- Université Libre de Bruxelles -
Faculté des sciences appliquées

Mémoire de fin d'études :

Génération automatique de code à partir des diagrammes d'état transition.

Année académique 2007/2008

Directeur de mémoire :
Prof. Hugues Bersini

Mémoire de fin d'études
présenté par **Vitale Pasquale**
en vue de l'obtention du grade
d'ingénieur civil informaticien.

Remerciements

En préambule de ce mémoire, je souhaite adresser ici tous mes remerciements aux personnes qui m'ont apporté leur aide et qui ont ainsi contribué à la réalisation de ce mémoire.
Je tiens tout d'abord à remercier Monsieur le professeur **Hugues Bersini**, directeur de ce mémoire pour son aide, ses conseils et la disponibilité qu'il a bien voulu m'accorder tout au long de l'année académique. Je voudrais aussi remercier mes parents, mes proches, mes amis pour leur soutien moral durant l'élaboration de ce mémoire.

Table des matières

Table des figures

Introduction

La génération automatique de code à partir des seuls diagrammes UML est la voie du future pour les développements logiciels. A ce jour, alors que les diagrammes de classe et de séquence ont déjà fait l'objet de beaucoup d'attention, il n'en va pas de même pour le diagramme d'état-transition.

Ce mémoire vise concrètement à implémenter un générateur de code automatique qui représentera le plus précisément possible un problème décrit uniquement par son diagramme d'état-transition.
Pour ce faire, il sera d'abord préférable de définir les notions utiles à la bonne compréhension de la démarche poursuivie. C'est pourquoi un premier chapitre sera consacré à l'explication des diagrammes UML et plus particulièrement du diagramme d'état-transition.
Au-delà des langages de programmation ou de modélisation (UML), la maîtrise des design patterns permet aux informaticiens d'attaquer la simulation de procédés complexes avec plus de facilité. Lors de ce mémoire l'un de ceux-ci sera largement utilisé : le State pattern, il sera donc nécessaire d'y consacrer un chapitre.
L'idée originale de ce mémoire sera d'utiliser ce pattern pour la simulation des diagrammes d'état transition. Nous présenterons un problème concret illustré par son diagramme que nous traduirons en code avant de passer à la génération automatique proprement dite. Un chapitre de ce mémoire sera d'ailleurs dédié à l'explication du procédé de génération de code automatique.
Après avoir assimilé ces différentes notions utiles, nous pourrons présenter les résultats obtenus et nous terminerons par une conclusion.

Première partie

Partie théorique

Chapitre 1

Les diagrammes UML

Pour programmer une application, il ne convient pas de se lancer tête baissée dans l'écriture du code : il faut d'abord organiser ses idées, les documenter, puis organiser la réalisation en définissant les modules et étapes de celle-ci. C'est cette démarche antérieure à l'écriture que l'on appelle *modélisation* ; son produit est *un modèle*.
Ce modèle représentera donc un ensemble d'éléments d'une partie du monde réel en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objets. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel ou virtuel (taille, couleur, ...) mais aussi de comportements (actions entre objets).La difficulté de cette modélisation consiste à créer une représentation abstraite, sous forme d'objets, d'entités ayant une existence matérielle (chien, voiture, ampoule, ...) ou bien virtuelle (sécurité sociale, temps, ...). Un exemple est fourni à la figure1.1. Il s'agit d'un objet voiture, on y retrouve certaines caractéristiques d'une voiture, ainsi que les différentes actions qu'il est possible de faire.

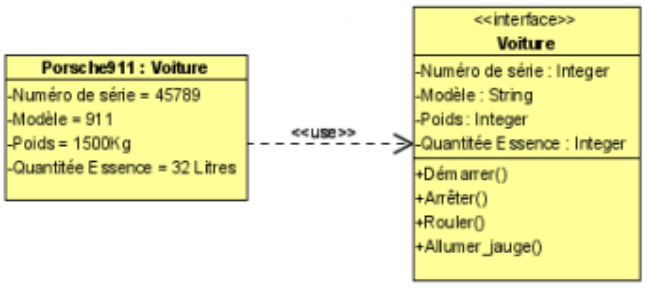

FIG. 1.1 – Exemple objet voiture

Les méthodes objet
La modélisation objet consiste à créer une représentation informatique des éléments du monde réel auxquels on s'intéresse, sans se préoccuper de l'implémentation, ce qui signifie indépendamment d'un langage de programmation. Il s'agit donc de déterminer les objets présents et d'isoler leurs données et les fonctions qui les utilisent. Pour se faire, des méthodes, des processus méthodologiques ont été mis au point. Au fil des années le nombre de ces méthodes n'a cessé de croître, si bien qu'en 1994 il existait plus de cinquante méthodes objets.

1.1 La naissance de l'UML

Parmi le nombre de ces méthodes, seulement trois se sont démarquées.

- La méthode OMT de Rumbaugh
- La méthode BOOCH 93 de Booch
- La méthode OOSE de Jacobson Object Oriented Software Engineering)

En 1994,Rumbaugh et Booch unissent leurs efforts pour mettre au point la méthode unifiée (Unified Method0.8), incorporant les avantages de chacune des méthodes précédentes. La méthode unifiée à partir de la version 1.0 devient UML (Unified Modeling Language), une notation universelle pour la modélisation objet.
UML est un moyen d'exprimer des modèles objet en faisant abstraction de leur implémentation. Le langage UML 2.0 comporte treize types de diagrammes. Citons les de façon hiérarchique :
Il y a les diagrammes de *structure (Structure diagrams)* qui mettent l'accent sur ce que les choses doivent être dans le système qui est en train d'être modélisé et qui regroupent :

- Les diagrammes de classes
- Les diagrammes de composants
- Les diagrammes de structures composites
- Les diagrammes de déploiement
- Les diagrammes d'objets
- Les diagrammes de paquets

ensuite il y a les diagrammes de *comportement (Behavior diagrams)* représentent ce qui se passe dans le système et qui regroupent :

- Les diagrammes d'activité
- Les diagrammes d'états-transitions
- Les diagrammes de cas d'utilisation

et finalement il y a les diagrammes d'*interaction (Interaction diagrams)* représentent les données échangées parmi les objets du système et qui regroupent :

- Les diagrammes de communication
- Les diagrammes globaux d'interaction
- Les diagrammes de séquence
- Les diagrammes de temps

Les types de diagrammes qui nous intéresseront tout au long de ce travail sont les diagrammes d'états-transitions. C'est pourquoi nous allons les détailler plus particulièrement dans la prochaine section.

1.2 Les diagrammes d'états-transitions

1.2.1 Introduction au formalisme

Présentation

Les diagrammes d'états-transitions d'UML décrivent le comportement interne d'un objet à l'aide d'un automate à états finis. Ils présentent les séquences possibles d'états et d'actions qu'une instance de classe peut traiter au cours de son cycle de vie en réaction à des événements discrets (de type signaux, invocations de méthode).
Ils spécifient habituellement le comportement d'une instance de classeur (classe ou composant), mais parfois aussi le comportement interne d'autres éléments tels que les cas d'utilisation, les sous-systèmes, les méthodes.
Le diagramme d'états-transitions est le seul diagramme, de la norme UML, à offrir une vision complète et non ambiguë de l'ensemble des comportements de l'élément auquel il est attaché. En effet, un diagramme d'interaction n'offre qu'une vue partielle correspondant à un scénario sans spécifier comment les différents scénarii interagissent entre eux.
La vision globale du système n'apparaît pas sur ce type de diagramme puisqu'ils

ne s'intéressent qu'à un seul élément du système indépendamment de son environnement.
Concrètement, un diagramme d'états-transitions est un graphe qui représente un automate à états finis, c'est-à-dire une machine dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées.

Notion d'automate à états finis

Comme nous venons de le dire, un automate à états finis est un automate dont le comportement des sorties ne dépend pas seulement de l'état de ses entrées, mais aussi d'un historique des sollicitations passées. Cet historique est caractérisé par *un état global.*
Un état global est un jeu de valeurs d'objet, pour une classe donnée, produisant la même réponse face aux événements. Toutes les instances d'une même classe ayant le même état global réagissent de la même manière à un événement. Il ne faut pas confondre les notions d'état global et d'état.
Un automate à états finis est graphiquement représenté par un graphe comportant des états, matérialisés par des rectangles aux coins arrondis, et des transitions, matérialisées par des arcs orientés liant les états entre eux.

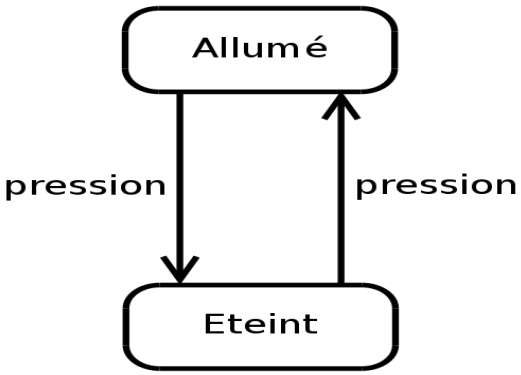

FIG. 1.2 – Un diagramme d'états-transitions simple

La figure 1.2 montre un exemple simple d'automate à états finis. Cet automate possède deux états (Allumé et Eteint) et deux transitions correspondant au même évènement : la pression sur un bouton d'éclairage domestique. Cet automate à états finis illustre en fait le fonctionnement d'un télérupteur dans une maison. Lorsque l'on appuie sur un bouton d'éclairage, la réaction de l'éclairage associé dépendra de son état courant (de son historique) : s'il la

lumière est allumée, elle s'éteindra, si elle est éteinte, elle s'allumera.

1.2.2 Un état

Présentation

Comme nous l'avons déjà dit, un état, que l'on peut qualifier informellement d'élémentaire, se représente graphiquement dans un diagramme d'états-transitions par un rectangle aux coins arrondis(voir figure 1.2). Certains états, dits composites (cf. section états composites plus après), peuvent contenir (i.e. envelopper) des sous-états. Le nom de l'état peut être spécifié dans le rectangle et doit être unique dans le diagramme d' états-transitions, ou dans l'état enveloppant. On peut l'omettre, ce qui produit un état anonyme. Il peut y avoir un nombre quelconque d'états anonymes distincts.
Un état peut être partitionné en plusieurs compartiments séparés par une ligne horizontale. Le premier compartiment contient le nom de l'état et les autres peuvent recevoir des transitions internes (cf. section transtions), ou des sous-états (cf. section états composites), quand il s'agit d'un état composite. Dans le cas d'un état simple (i.e. sans transitions interne ou sous-état), on peut omettre toute barre de séparation.

Etat initial et état final

1. **Etat initial**

   L'état initial est un pseudo état qui indique l'état de départ, par défaut, lorsque le diagramme d'états-transitions, ou l'état enveloppant, est invoqué. Lorsqu'un objet est créé, il entre dans l'état initial.

FIG. 1.3 – Exemple d'état initial

2. **Etat final**

   L'état final est un pseudo état qui indique que le diagramme d'états-transitions, ou l'état enveloppant, est terminé.

FIG. 1.4 – Exemple d'état final

1.2.3 Les événements

Notion

Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé. Les diagrammes d'états-transitions permettent justement de spécifier les réactions d'une partie du système à des événements discrets. Un événement se produit à un instant précis et est dépourvu de durée. Quand un événement est reçu, une transition peut être déclenchée et faire basculer l'objet dans un nouvel état. On peut diviser les événements en plusieurs types explicites et implicites : signal, appel, changement et temporel.

Les signaux : Un signal est un type destiné explicitement à véhiculer une communication asynchrone à sens unique entre deux objets. L'objet expéditeur crée et initialise explicitement une instance de signal et l'envoie à un objet explicite ou à tout un groupe d'objets. Il n'attend pas que le destinataire traite le signal pour poursuivre son déroulement. La réception d'un signal est un événement pour l'objet destinataire. Un même objet peut être à la fois expéditeur et destinataire.

Les appels : Un événement d'appel représente la réception de l'appel d'une opération par un objet. Les paramètres de l'opération sont ceux de l'événement d'appel.

Les changements : Un événement de changement est généré par la satisfaction (i.e. passage de faux à vrai) d'une expression booléenne sur des valeurs d'attributs. Il s'agit d'une manière déclarative d'attendre qu'une condition soit satisfaite.

Les événements temporels : Les événements temporels sont générés par le passage du temps. Ils sont spécifiés soit de manière absolue (date précise), soit de manière relative (temps écoulé). Par défaut, le temps commence à s'écouler dès l'entrée dans l'état courant.

1.2.4 Les transitions

Définition et syntaxe

Une transition définit la réponse d'un objet à l'occurrence d'un événement. Elle lie, généralement, deux états E1 et E2 et indique qu'un objet dans un état E1 peut entrer dans l'état E2 et exécuter certaines activités, si un événement déclencheur se produit et que la condition de garde est vérifiée. La syntaxe d'une transition est la suivante :

<événement> [<garde>] /<activité>

Le même événement peut être le déclencheur de plusieurs transitions quittant un même état. Chaque transition avec le même événement doit avoir une condition de garde différente. En effet, une seule transition peut se déclencher dans un même flot d'exécution. Si deux transitions sont activées en même temps par un même événement, une seule se déclenche et le choix n'est pas prévisible (i.e. pas déterministe).

Les conditions de garde

Une transition peut avoir une condition de garde (spécifiée par [<garde>] dans la syntaxe). Il s'agit d'une expression logique sur les attributs de l'objet associé au diagramme d'états-transitions ainsi que sur les paramètres de l'événement déclencheur. La condition de garde est évaluée uniquement lorsque l'événement déclencheur se produit. Si l'expression est fausse à ce moment là, la transition ne se déclenche pas, si elle est vraie, la transition se déclenche et ses effets se produisent.

Effet d'une transition

Lorsqu'une transition se déclenche, son effet (spécifié par /<activité> dans la syntaxe) s'exécute. Il s'agit généralement d'une activité qui peut être :

- une opération primitive comme une instruction d'assignation ;
- l'envoi d'un signal ;
- l'appel d'une opération ;
- une liste d'activités.

La façon de spécifier l'activité à réaliser est laissée libre (langage naturel ou pseudo-code). Lorsque l'exécution de l'effet est terminée, l'état cible de la transition devient actif.

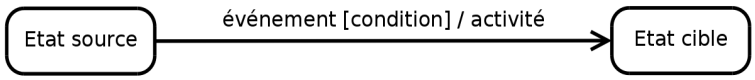

FIG. 1.5 – Représentation graphique d'une transition

La figure 1.5 illustre la représentation graphique d'une transition entre deux états, utilisant les notions définies.

Les transitions internes

Les règles de déclenchement d'une transition interne sont les mêmes que pour les transitions définies plus haut excepté qu'une transition interne ne possède pas d'état cible et que l'état actif reste le même à la suite de son déclenchement. La syntaxe d'une transition interne reste la même que celle d'une transition classique. Par contre, les transitions internes ne sont pas représentées par des arcs mais sont spécifiées dans un compartiment de leur état associé.

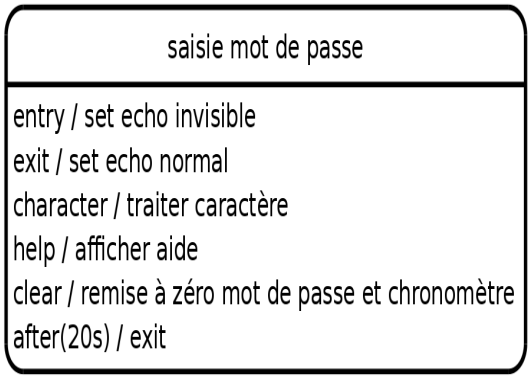

FIG. 1.6 – Représentation graphique d'une transition interne

Les transitions internes possèdent des noms d'événements prédéfinis correspondants à des déclencheurs particuliers : **entry, exit, do et include**. Ces mots clefs réservés viennent prendre la place du nom de l'événement dans la syntaxe d'une transition interne.

entry : permet de spécifier une activité qui s'accomplit quand on entre dans l'état.

exit : exit permet de spécifier une activité qui s'accomplit quand on sort de l'état.

do : Une activité do commence dès que l'activité entry est terminée. Si une transition se déclenche pendant que l'activité do est en cours, cette dernière est interrompue et l'activité exit de l'état s'exécute.

include : permet d'invoquer un sous-diagramme d'états-transitions.

Les activités entry servent souvent à effectuer la configuration nécessaire dans un état. Comme il n'est pas possible de l'éluder, toute action interne à l'état peut supposer que la configuration est effectuée indépendamment de la manière dont on entre dans l'état. De manière analogue, une activité exit est une occasion de procéder à un nettoyage. Cela peut s'avérer particulièrement utile lorsqu'il existe des transitions de haut niveau qui représentent des conditions d'erreur qui abandonnent les états imbriqués.
Le déclenchement d'une transition interne ne modifie pas l'état actif et n'entraîne donc pas l'activation des activités entry et exit.

1.2.5 Les points de choix

Il est possible de représenter des alternatives pour le franchissement d'une transition. On utilise pour cela des pseudo-états particuliers : les points de jonction (représentés par un petit cercle plein) et les points de décision (représenté par un losange).

Les points de jonction

Les points de jonction sont un artefact graphique (un pseudo-état en l'occurrence) qui permet de partager des segments de transition, l'objectif étant d'aboutir à une notation plus compacte ou plus lisible des chemins alternatifs.
Un point de jonction peut avoir plusieurs segments de transition entrante et plusieurs segments de transition sortante. Par contre, il ne peut avoir d'activité interne ni des transitions sortantes dotées de déclencheurs d'événements. Il ne s'agit pas d'un état qui peut être actif au cours d'un laps de temps fini. Lorsqu'un chemin passant par un point de jonction est emprunté (donc lorsque la transition associée est déclenchée) toutes les gardes le long de ce chemin doivent s'évaluer à vrai dès le franchissement du premier segment.

La figure 1.6 montre un diagramme sans point de jonction. La figure 1.7 montre son équivalent utilisant un tel point.

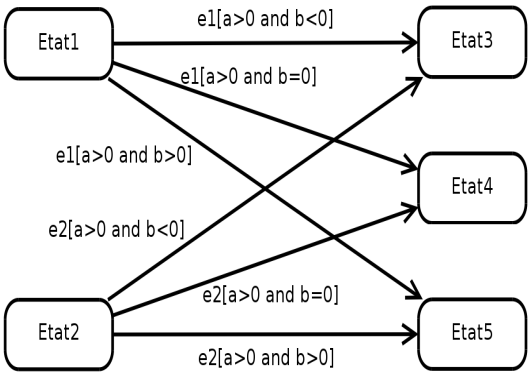

FIG. 1.7 – Représentation graphique d'un diagramme sans points de jonction

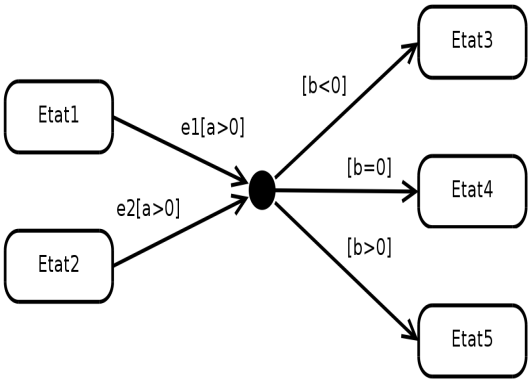

FIG. 1.8 – Représentation graphique d'un diagramme avec points de jonction

Les points de décision

Un point de décision possède une entrée et au moins deux sorties. Contrairement à un point de jonction, les gardes situées après le point de décision sont évaluées au moment où il est atteint. Cela permet de baser le choix sur des résultats obtenus en franchissant le segment avant le point de choix.
Si, quand le point de décision est atteint, aucun segment en aval n'est franchissable, c'est que le modèle est mal formé.
Il est possible d'utiliser une garde particulière, notée [else], sur un des segments en aval d'un point de choix. Ce segment n'est franchissable que si les gardes des autres segments sont toutes fausses.
L'utilisation d'une clause [else] est recommandée après un point de décision car elle garantit un modèle bien formé. (cfr figure 1.9)

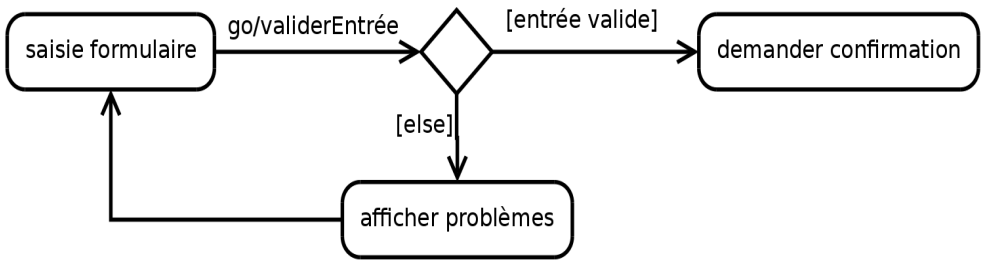

FIG. 1.9 – Représentation graphique d'un diagramme avec point de décision

La figure 1.9 illustre l'utilisation d'un point de décision.

1.2.6 Les états composites

Définition

Un état simple ne possède pas de sous-structure mais uniquement, le cas échéant, un jeu de transitions internes. Un état composite est un état décomposé en régions contenant chacune un ou plusieurs sous-états.
Quand un état composite comporte plus d'une région, il est qualifié d'état orthogonal. Lorsqu'un état orthogonal est actif, un sous-état direct de chaque région est simultanément actif. Un état composite ne comportant qu'une région est qualifié d'état non orthogonal.
Implicitement, tout diagramme d'états-transitions est contenu dans un état externe qui n'est usuellement pas représenté. Cela apporte une plus grande homogénéité dans la description : **tout diagramme d'états-transitions est implicitement un état composite.**

L'utilisation d'états composites permet de développer une spécification par raffinements. Il n'est pas nécessaire de représenter les sous-états à chaque utilisation de l'état englobant. Une notation abrégée permet d'indiquer qu'un état est composite et que sa définition est donnée sur un autre diagramme.

La figure figure 1.10 montre un exemple d'état composite et la figure 1.11 montre sa notation abrégée.

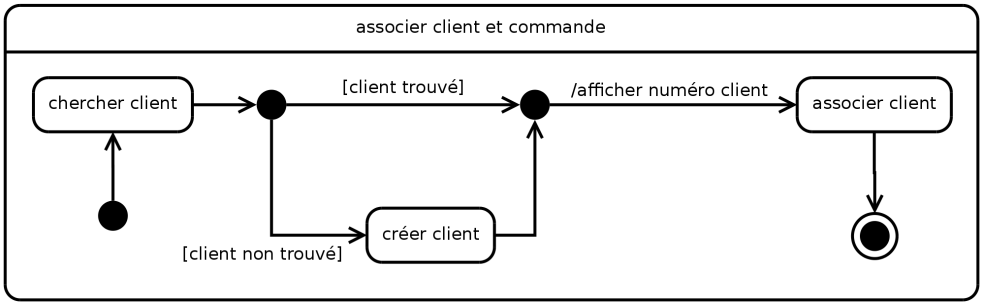

FIG. 1.10 – Exemple d'état composite modélisant l'association d'une commande à un client.

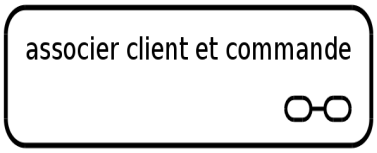

FIG. 1.11 – Notation abrégée d'un état composite.

Les transitions

Les transitions peuvent avoir pour cible la frontière d'un état composite et sont équivalentes à une transition ayant pour cible l'état initial de l'état composite.
Une transition ayant pour source la frontière d'un état composite est équivalente à une transition qui s'applique à tout sous-état de l'état composite source. Cette relation est transitive : la transition est franchissable depuis tout état imbriqué, quelle que soit sa profondeur.
Par contre, si la transition ayant pour source la frontière d'un état composite ne porte pas de déclencheur explicite (i.e. s'il s'agit d'une transition d'achèvement), elle est franchissable quand l'état final de l'état composite est atteint.

Les transitions peuvent également toucher des états de différents niveaux d'imbrication en traversant les frontières des états composites.

Sur la figure 1.12 nous voyons que depuis l'état État 1, la réception de l'événement event1 produit la séquence d'activités *QuitterE11, QuitterE1, action1, EntrerE2, EntrerE21, initialiser(), EntrerE22,* et place le système dans l'état État22.

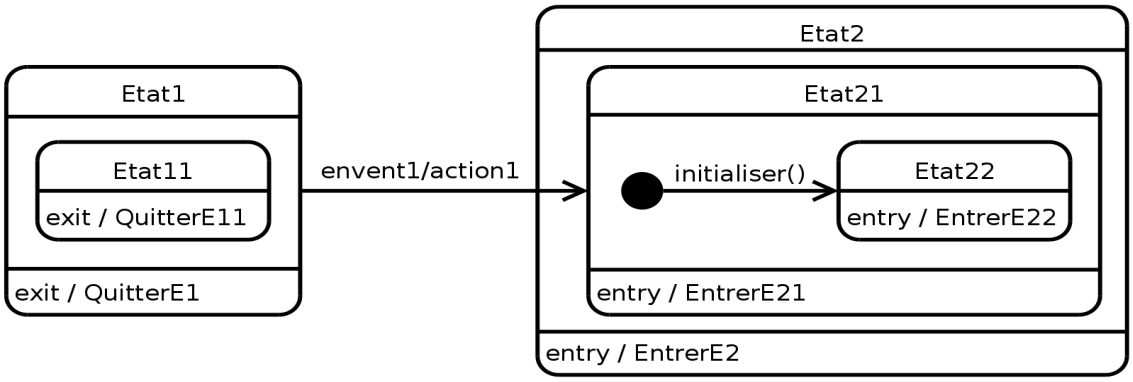

FIG. 1.12 – Exemple de configuration complexe de transitions

L'état historique

Un état historique, également qualifié d'état historique plat, est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Graphiquement, il est représenté par un cercle contenant un H.

Une transition ayant pour cible l'état historique est équivalente à une transition qui a pour cible le dernier état visité de l'état englobant. Un état historique peut avoir une transition sortante non étiquetée indiquant l'état à exécuter si la région n'a pas encore été visitée.

Il est également possible de définir un état historique profond représenté graphiquement par un cercle contenant un H*. Cet état historique profond permet d'atteindre le dernier état visité dans la région, quel que soit son niveau d'imbrication, alors que le l'état historique plat limite l'accès aux états de son niveau d'imbrication.

La figure 1.13 montre un diagramme d'états-transitions modélisant le lavage automatique d'une voiture. Les états de lavage, séchage et lustrage sont des états composites définis sur trois autres diagrammes d'états-transitions non représentés ici. En phase de lavage ou de séchage, le client peut appuyer sur le bouton d'arrêt d'urgence. S'il appuie sur ce bouton, la machine se met

en attente. Il a alors deux minutes pour reprendre le lavage ou le lustrage, exactement où le programme a été interrompu, c'est- à-dire au niveau du dernier sous-état actif des états de lavage ou de lustrage (état historique profond). Si l'état avait été un état historique plat, c'est toute la séquence de lavage ou de lustrage qui aurait été recommencée. En phase de lustrage, le client peut aussi interrompre la machine. Mais dans ce cas, la machine s'arrêtera définitivement.

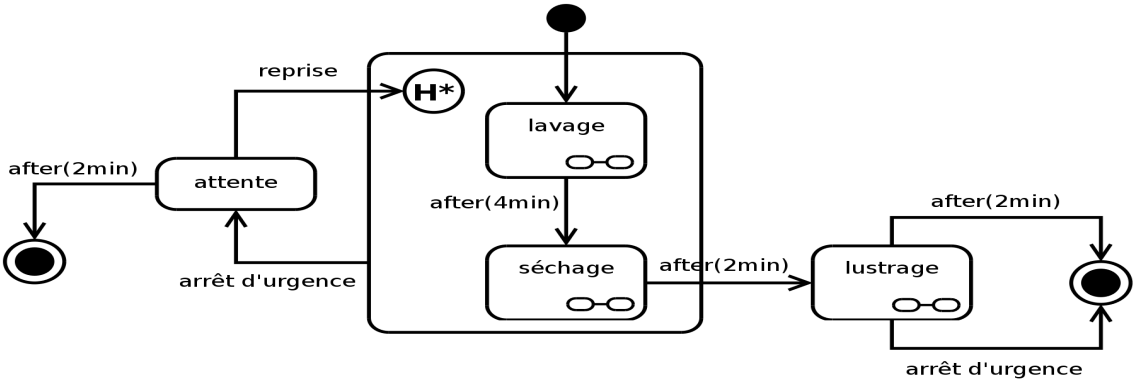

FIG. 1.13 – Exemple de diagramme possédant un état historique profond

Interface : Les points de connexions

Comme nous l'avons déjà dit, il est possible de masquer les sous-états d'un état composite et de les définir dans un autre diagramme. Cette pratique nécessite parfois l'utilisation de pseudo-états appelés points de connexion.

Lorsque l'on utilise le comportement par défaut de l'état composite, c'est-à-dire entrer par l'état initial par défaut et considérer les traitements finis quand l'état final est atteint, aucun problème ne se pose car on utilise des transitions ayant pour cible, ou pour source, la frontière de l'état composite. Dans ce cas, les points de connexion sont inutiles.

Le problème se pose lorsqu'il est possible d'entrer ou de sortir d'un état composite de plusieurs façons. C'est, par exemple, le cas lorsqu'il existe des transitions traversant la frontière de l'état composite et visant directement, ou ayant pour source, un sous-état de l'état composite. Dans ce cas, la solution est d'utiliser des points de connexion sur la frontière de l'état composite.

Les points de connexion sont des points d'entrée et de sortie portant un nom, et situés sur la frontière d'un état composite. Ils sont respectivement représentés par un cercle vide et un cercle barré d'une croix. Il ne s'agit que de références à un état défini dans l'état composite. Une unique transition d'achèvement, dépourvue de garde, relie le pseudo-état source (i.e. le point de connexion) à l'état référencé. Cette transition d'achèvement n'est que le prolongement de la transition qui vise le point de connexion (il peut d'ailleurs y en avoir plusieurs). Les points de connexions offrent ainsi une façon de représenter l'interface (au sens objet) d'un état composite en masquant l'implémentation de son comportement.

On peut considérer que les pseudo-états initiaux et finaux sont des points de connexion sans nom.

La figure 1.14 illustre l'utilisation de points de connexions.

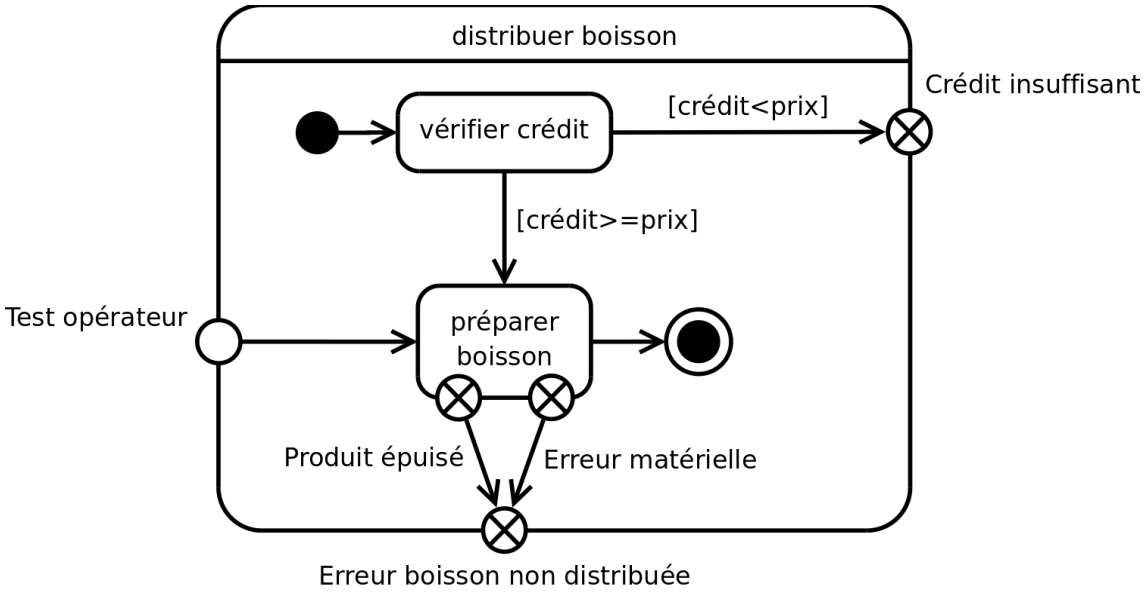

FIG. 1.14 – Exemple d'utilisation de points de connexion.

1.3 Conclusion

Nous venons de définir en détails les diagrammes d'états-transitions.
La prochaine étape de la démarche sera de définir les notions utiles à la bonne compréhension du design pattern : *Le pattern State*, afin de pouvoir établir un diagramme d'états-transitions complet et de pouvoir transcrire celui-ci grâce au pattern State.

Chapitre 2

Les design patterns

Un design pattern est un concept destiné à résoudre des problèmes récurrents suivant le paradigme objet. Ils décrivent des solutions standards pour répondre à des problèmes d'architecture et de conception des logiciels. À la différence d'un algorithme qui s'attache à décrire d'une manière formelle comment résoudre un problème particulier, les patterns décrivent des procédés de conception généraux.
Il ne s'agit pas de fragments de code, puisque ceux-ci sont le plus souvent indépendants du langage de programmation, mais d'une méthode de conception, c'est-à-dire d'une manière standardisée de résoudre un problème qui s'est déjà posé par le passé. Le concept de design patterns a donc une grande influence sur l'architecture logicielle d'un système.

2.1 Historique

L'origine des design patterns remonte au début des années 70 avec les travaux de l'architecte Christopher Alexander et avec son livre "A pattern language".[1] Celui-ci remarque que la phase de conception en architecture laisse apparaître des problèmes récurrents. Il dira d'ailleur dans ce livre :
" *Chaque pattern décrit un problème récurrent dans notre environnement, ainsi que ce qui constitue la noyau de la solution à ce problème d'une telle façon que vous pouvez utilisez cette solution un million de fois sans jamais la faire 2 fois de la même façon".*
Dans les années 90, l'idée de Christopher Alexander va être reprise et étendue au domaine de la conception des logiciels. Le concept de Design Pattern est développé dans un ouvrage publié en 1995 par le "Gang of Four". Cette équipe qui regroupe Erich Gamma, Richard Helm, Ralph Johnson et John

[1] "A pattern language";C.Alexander;New-York : Oxford university press; 1977

Vlissides intitule ce livre "Design Patterns : Elements of Reusable Object-Oriented Software". Celui-ci présente 23 Design Patterns qui font aujourd'hui référence dans le monde de l'informatique.

2.2 Description

On distingue trois familles de design patterns selon leur utilisation :

créationnels : ils définissent comment faire l'instanciation et la configuration des classes et des objets.

structuraux : ils définissent comment organiser les classes d'un programme dans une structure plus large (séparant l'interface de l'implémentation).

comportementaux : ils définissent comment organiser les objets pour que ceux-ci collaborent (distribution des responsabilités) et expliquent le fonctionnement des algorithmes impliqués.

Nous nous contenterons ici de citer les différents patterns les plus connus sans rentrer dans les détails car cela dépasserait l'objectif de ce mémoire. Nous consacrerons tout de même un chapitre particulier à l'explication détaillée du State pattern qui nous sera très utile pour la compréhension de la suite de la démarche.

Créationnels	**Structuraux**	**Comportementaux**
Abstract Factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

FIG. 2.1 – Liste des design patterns

2.3 Le pattern State

Expliquons le formalisme en plusieurs étapes :

Le problème :

Le comportement d'un objet dépend de son état.

→ La plupart des personnes utiliseraient les conditions logiques pour palier à ce problème. C'est-à-dire tester l'appartenance d'un objet à un certain état par une série de conditions " if " informatique et changer d'état, le cas échéant, à ce moment là. Mais cette solution n'est guère souhaitable car trop complexe. De plus elle introduit un certain facteur de duplication de code qui n'est pas optimal.

La solution :

- Créer une classe pour chaque état possible pouvant être pris par l'objet en question.
- Assigner une méthode, de manière polymorphique, à chaque classe d'état permettant de traiter le comportement de l'objet.
- Quand l'objet reçoit un message de changement de comportement ; délégué ce changement à la classe état correspondant.

La définition

"*Le pattern état permet à un objet de modifier son comportement quand son état interne change. Tout se passera comme si l'objet changeait de classe.*"

→ La première partie de la définition est assez claire. Par contre il est peut être utile de parler un peu de la deuxième.
Que veut dire "*tout se passera comme si l'objet changeait de classe*" ?
Mettons-nous à la place d'un utilisateur. Si un objet que nous utilisons change totalement de comportement, nous pouvons alors penser tout naturellement qu'il est instancié à partir d'une autre classe. Mais, en réalité, vous savez que nous utilisons une composition pour donner l'apparence d'un changement de classe en référençant simplement des objets état différent.
Passons au diagramme de classe de ce pattern pour plus de compréhension.

Le diagramme de classe :

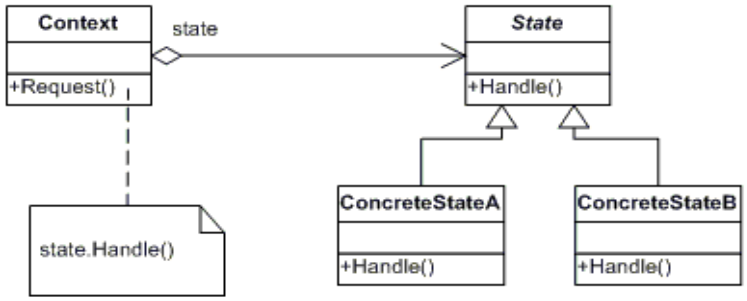

FIG. 2.2 – Diagramme de classe du State pattern

Explication de la figure :

Le Context : est la classe qui peut avoir plusieurs états internes.

state.Handle() : Chaque fois que la requête est effectuée sur le Context, sa gestion est déléguée à l'état.

L'interface State : définit une interface commune pour tous les états concrets.

Les états concrets : chacun de ceux-ci gère les requêtes du Context et fournit sa propre implémentation de la requête. De cette manière, quand le Context change d'état, son comportement change également.

2.4 Conclusion

Maintenant que nous avons assimilé toute la théorie nécessaire à la compréhension du mémoire, nous allons pouvoir commencer à la mettre en pratique. Pour cela nous allons nous intéresser à un problème concret, que nous expliquerons en détails, avant de le traduire en diagramme et en code.

Deuxième partie

Partie pratique

Chapitre 3

La mise en pratique

Dans un premier temps, nous allons poser le problème. Ensuite nous traduirons celui-ci en son équivalent UML, c'est-à-dire en diagramme d'états transitions.
Comme nous l'avons déjà dit plus avant, l'idée originale de ce travail sera d'utiliser le State pattern pour traduire ce diagramme UML en lignes de code. Ce sera donc l'étape suivante de la démarche.

3.1 Posons le problème

Nous allons nous inspirer du problème décrit dans le livre sur les design patterns que nous avons utilisé lors de ce mémoire.[1]
Celui-ci décrit la demande d'une certaine entreprise de fabrication de distributeurs de bonbons de pouvoir implémenter le fonctionnement d'un de ceux-ci d'une manière aussi souple et facile que possible. Et ce, afin de pouvoir, le cas échéant, rajouter d'autres spécifications au distributeur en question.
Les ingénieurs de cette entreprise ont soumis ce problème sous forme d'un pseudo diagramme d'états-transitions.(cfr page suivante)

[1] "Design patterns - Tête la première" ; Er.Freeman, El.Freeman, K.Sierra, B.Bates ; ed. O'Reilly, Paris 2005

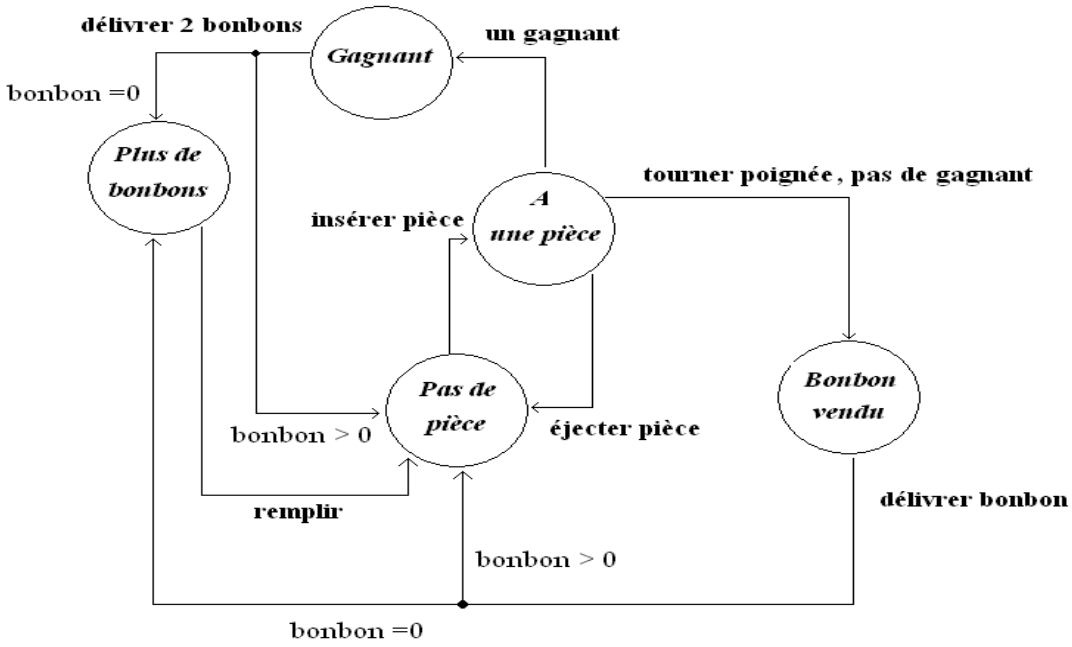

FIG. 3.1 – Fonctionnement du distributeur de bonbon

Explication de la figure

Nous pouvons voir sur cette figure les différents états par lesquels peut passer notre distributeur (ce sont les cercles de la figure), à savoir : plus de bonbon, gagnant, à une pièce, pas de pièce et bonbon vendu. Ces différents états sont reliés entre eux par des transitions déclanchées par certains événements (insérer pièce, retirer pièce, ...).

Le distributeur possède une petite particularité. En effet, nous pouvons voir qu' à partir de l'état " A pièce " soit la machine vend un bonbon et passe à l'état " Bonbon vendu ", soit la machine passe dans un état " Gagnant " et délivre alors, en guise de cadeau, un deuxième bonbon pour la même pièce injectée dans l'appareil.

Nous pouvons apercevoir aussi 2 flèches qui prennent une direction différente suivant le nombre de bonbon restant dans le distributeur. Ce cas précis peut se traduire avec l'utilisation de points de décision (cfr chapitre 1).

Traduisons donc ce pseudo diagramme à l'aide des notions que nous avons acquises lors des premiers chapitres de ce travail.

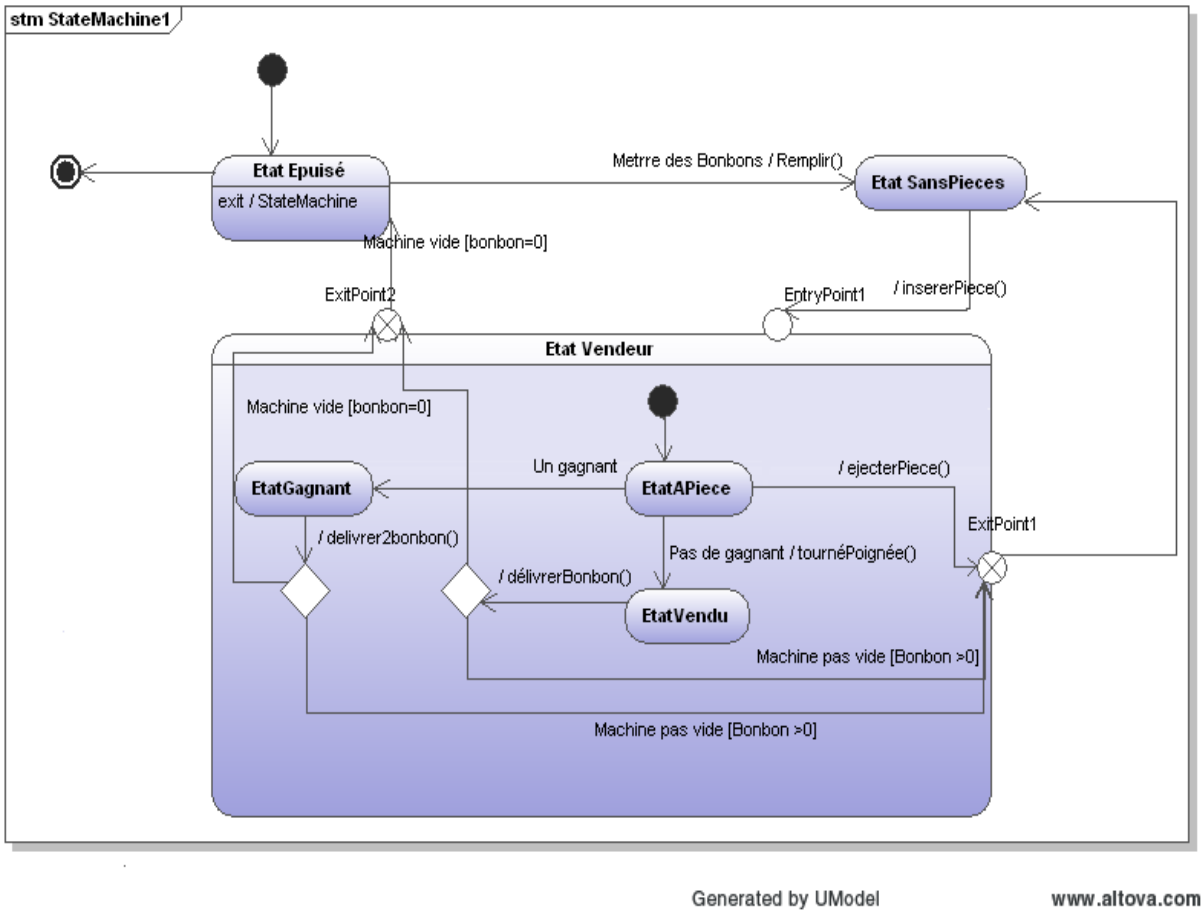

FIG. 3.2 – Diagramme d'états-transitions du distributeur de bonbons.

Explication du diagramme :
Nous voyons que l'état initial de la machine est l'état Epuisé, il est d'ailleurs aussi l'état final. Nous avons ajouté une petite spécification au diagramme en ce sens qu'on y trouve un état composite qui porte le nom d'état Vendeur. Celui-ci a été ajouté pour diverse raison. D'une part il pourrait faciliter la compréhension du fonctionnement de la machine si nous l'utilisions sous sa notation abrégée (cfr chapitre 1). D'autre part, il améliore la pédagogie des résultats car le diagramme utilise alors la plupart des notions vues lors de la partie théorique de ce travail.
L'état initial de cet état composite est l'état à pièce, c'est-à-dire que ce sera le premier état atteint lorsque nous atteindrons l'état composite.

Nous pouvons aussi apercevoir les points de décision suite à la vérification du nombre de bonbon restant dans l'appareil, points qui aboutissent chacun vers des sorties différentes de l'état composite, Etat vendeur, sous formes de points de sortie.
Comme déjà dit précédemment, les états sont tous reliés entre eux par des transitions, écrites cette fois ci sous forme de méthodes informatiques.

3.2 Traduction du diagramme avec le pattern State

Avant de passer à la traduction proprement dite du diagramme, il est nécessaire d'apporter une brève explication de la manière dont nous allons procéder. Dans la majeure partie de cette étape nous allons utiliser le pattern State, comme nous l'avons déjà dit plusieurs fois auparavant. Cependant, nous allons en plus utiliser un autre design pattern et ce dans le but de simplifier et d'améliorer la validité de notre conception.
Nous avons remarqué dans la section précédente, qu'un nouvel état est apparu dans le diagramme. Nous parlons de l'état composite : état vendeur. C'est un état spécial car il contient lui-même une série de sous états, qui font de lui une sorte de macro état. Cette spécificité peut s'avérer être une difficulté lors du passage à la programmation de celui-ci. C'est pour cela que nous allons utiliser le design pattern : **Composite**. Nous ne l'avons pas détaillé particulièrement lors du second chapitre de ce mémoire pour ne pas trop nous encombrer l'esprit. C'est pourquoi nous allons ici nous contenter d'en donner sa définition et son équivalent en diagramme de classe, juste pour pouvoir le repérer et le comprendre dans le diagramme de classe général du distributeur de bonbon.

3.2.1 Le pattern Composite

Définition :

Le pattern composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même façon les objets individuels et les compositions de ceux-ci.

Diagramme de classe :

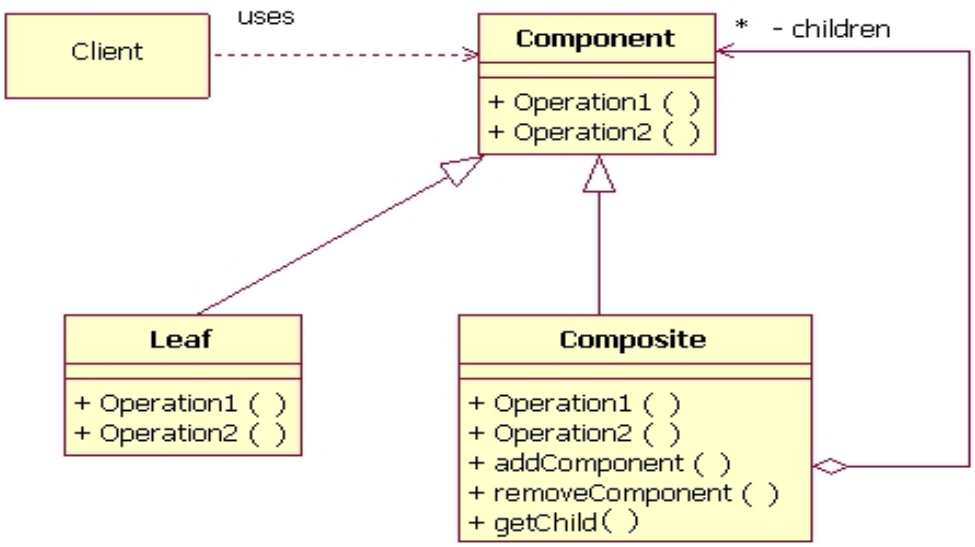

FIG. 3.3 – Diagramme de classe du composite pattern.

Nous y voyons qu'un client peut utiliser les méthodes opérations d'un component, indifféremment de savoir si l'objet pour lequel il appelle l'une de ces méthodes est une feuille (leaf) ou un composite. Nous voyons de plus qu'un composite peut être composé soit d'un composite même ou soit de component (qui sont soit des composites soit des feuilles).

3.2.2 Le diagramme de classe du distributeur

Nous sommes maintenant prêt à produire le diagramme de classes complet de notre distributeur. Après l'élaboration de celui-ci, la traduction en code du distributeur n'en sera que facilitée.

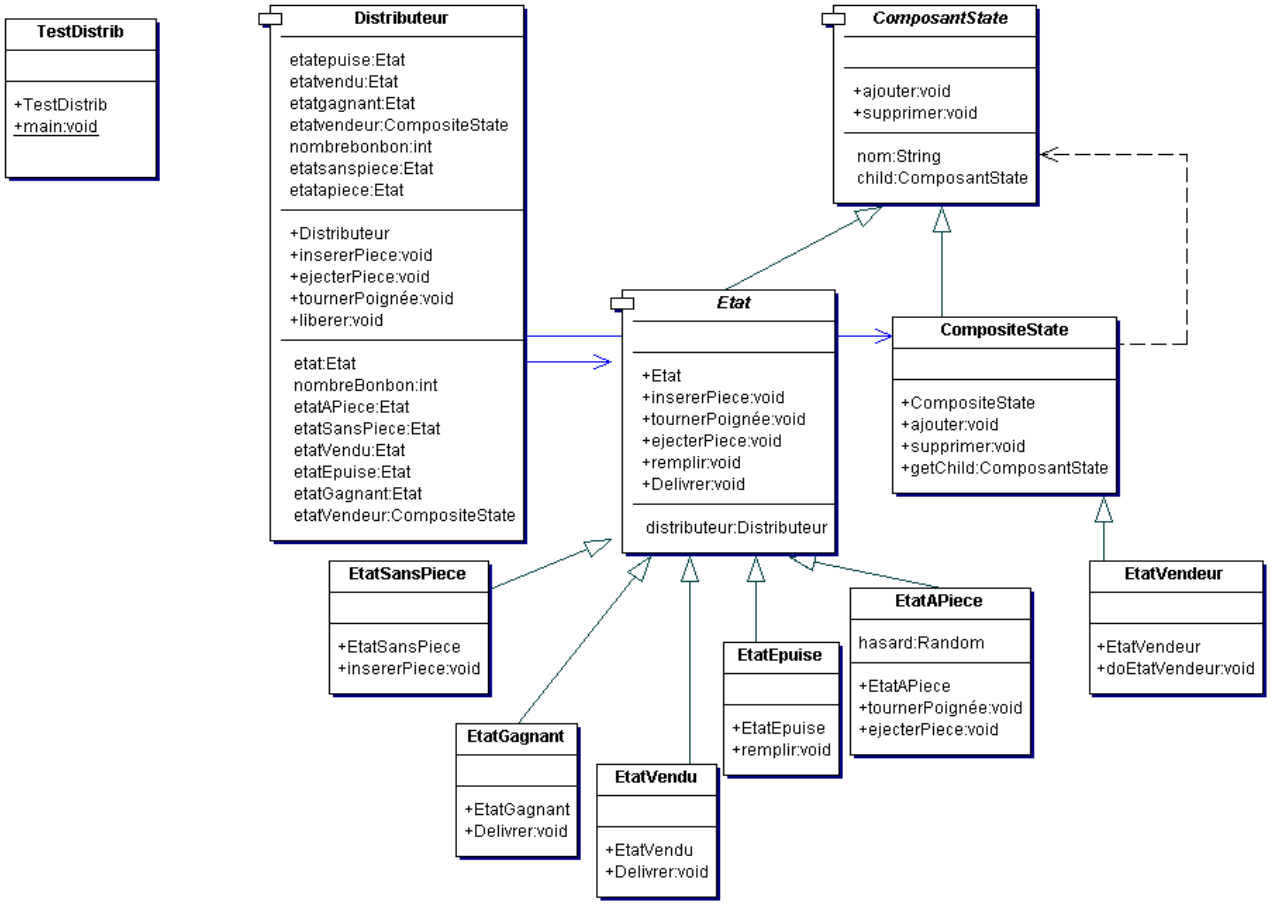

FIG. 3.4 – Diagramme de classe complet du distributeur.

Description du diagramme :

Comme nous pouvons le remarquer, le pattern State se retrouve dans ce diagramme. En effet le contexte d'exécution n'est autre que la classe " Distributeur " qui est associée avec la classe " Etat ". Celle-ci possède autant de sous classes qu'il n'y a d'états pour le distributeur. Chacune de ces sous classes possèdent leurs méthodes propres qui sont, rappelons-le, les transitions capables de modifier le comportement de l'appareil. Et chacune de ces méthodes se retrouve définie (mais non implémentée) dans la super classe Etat. Le distributeur n'a donc plus qu'à définir ces méthodes dans sa propre classe pour pouvoir les appeler sur un objet état qui représente l'état courant du distributeur. (Nous pouvons voir cet objet état dans la dernière partie de la classe Distributeur du diagramme).
Dans une seconde partie du diagramme, nous pouvons apercevoir la struc-

ture du pattern composite. Comme nous l'avons déjà dit plus haut, celle-ci a été ajoutée pour permettre de tenir compte des états composites. En effet, nous voyons que les états " simples " font office de feuilles au sens de la définition du pattern et que les états composites implémenteront la classe CompositeState comme le fait état " Vendeur ". Les classes compositeState et Etat héritent toutes deux de la super classe ComposantState qui fait office de Component au sens de la définition décrite à la section précédente.

3.3 Le code du distributeur

Nous allons dans cette section décrire brièvement les grandes lignes du code du distributeur. Nous disons brièvement car l'essentiel pour le moment est d'avoir bien compris les diagrammes de classe et d'états transitions pour pouvoir comprendre le fonctionnement interne du distributeur. Toutefois, il est nécessaire d'apporter un mot d'explication au code car les étapes suivantes de la démarche seront de générer du code et donc il est essentiel de pouvoir avoir un support de comparaison entre le code écrit (c'est-à-dire le code issus du diagramme de classe) et le code générer (c'est-à-dire celui issus du diagramme d'états transitions).

Avant de commencer l'explication, nous allons rapidement justifier notre choix du langage de programmation pour ce mémoire.
Il existe beaucoup de langages permettant la programmation orientée objet comme C++, phyton, smalltalk, java...
Nous avons choisi de programmer en langage JAVA pour plusieurs raisons :

- Java permet de s'affranchir des problèmes de gestion de mémoire grâce à son " Garbage Collector " contrairement à c++
- Phyton est langage plus récent que java, ce qui fait qu'une plus grande maîtrise de ce dernier langage est déjà acquise grâce aux divers travaux réalise précédemment.

3.3.1 Description du code

La classe Distributeur

Cette classe désigne donc le contexte d'exécution du problème. Elle crée dans son constructeur toutes les nouvelles instances des différents états possibles de la machine et elle crée une méthode différente pour chaque transition pouvant faire changer le comportement interne du distributeur. Ce sont ces méthodes qui méritent un mot d'explication pour la compréhension générale

du fonctionnement. Au lieu de toutes les insérer les unes à la suite des autres et de les décrire une par une, proposons nous plutôt de n'en insérer qu'une seule et de la décrire car les autres sont écrites suivant la même structure et cela permettra de ne pas encombrer le rapport.
Voici donc l'implémentation de la méthode *ejecterpiece()* qui est la méthode sensée être appelée à partir de l'état Sanspieces et sensée désigner le fait que le distributeur a ejecté la pièce qu'il contenait :

```
public void ejecterPiece(){
    String test= "ejecterPiece";
    Class EtatAPiece=null;
    EtatAPiece= etatapiece.getClass();
    Method[] methods = EtatAPiece.getDeclaredMethods();
    for (int i = 0; i < methods.length; i++){
        if(methods[i].getName().equals(test)){
         etat.ejecterPiece();
         System.out.println("l'etat modif est"+etat);
       }
     }
   };
```

FIG. 3.5 – La méthode ejecterpiece()

Nous y voyons bien que avant de pouvoir déléguer le changement de comportement à la classe état par la commande " etat.ejecterpiece() " il est nécessaire de faire un petit test.
En effet, il ne convient de pouvoir appeler cette fonction de la classe état que si nous nous trouvons dans état Etat à pièces comme le suggère d'ailleurs le diagramme d'états transitions.
Comme le distributeur ne peut pas savoir quel est son état interne avant l'exécution du code, il ne suffit pas de simplement tester la valeur de la variable Etat qui désigne justement l'état interne du distributeur. Donc pour palier à ce problème nous avons utilisé l'introspection de classe qui permet à une méthode de récupérer les noms de méthode d'une certaine classe. Ce faisant, comme nous savons qu'il n'existe qu'une et une seule méthode ejecterpiece, nous testons si cette méthode fait bien partie de la classe Etat a pièce et au tel cas nous permettons le changement de comportement. Nous opérons de la même manière pour les autres méthodes.

3.4 Conclusion

Nous pouvons maintenant considérer que nous avons terminé l'implémentation du distributeur de bonbon. [2] Nous sommes en possession des différents diagrammes UML nécessaires à la génération du code. Nous pouvons donc passer à celle-ci, ce sera l'objet des prochaines sections de ce rapport.

[2] Le lecteur intéressé pourra trouver en annexe les références pour accéder au code complet.

Chapitre 4

La génération de code

Nous allons expliquer dans cette section la manière de générer du code à partir du diagramme d'état transition. Dans un premier temps il s'agira de passer d'un diagramme à un fichier de données représentant celui-ci. Ensuite, à partir de ce fichier nous pourrons aller puiser les données nécessaires à la génération du code.

4.1 Le XMI

Dans le cadre de ce mémoire nous allons utiliser un standard basé sur XML car les outils de modélisations échangent des fichiers basés sur XML. Ce standard s'appelle le XMI.
XMI (XML Metadata Interchange) est une forme encapsulée de tags XML. Il est un standard créé par l'OMG [1] pour l'échange d'informations de métadonnées [2] UML basé sur XML. Il est le plus souvent utilisé comme standard d'échange entre différents outils de modélisation. Impossible de parler de XMI sans évoquer les deux autres standards qui lui sont intimement liés : UML et MOF. Nous connaissons maintenant bien l'UML grâce à la première partie de ce rapport, mais nous connaissons moins le 2 ème standard.
MOF (Meta Object Facility) utilise un sous-ensemble de UML pour décrire les objets manipulés par les outils de conception. Enfin, XMI (XML Metadata Interchange) indique comment les modèles MOF peuvent être traduits en XML. Le but de ces standards est de permettre à des ateliers logiciels d'

[1] Object Management Group est une association américaine à but non-lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

[2] Une métadonnée est une donnée servant à définir ou décrire une autre donnée quel que soit son support (papier ou électronique).

explorer et d'échanger les définitions des structures de données, leurs propriétés, les relations les unissant, etc. Nous allons, comme dit précédement, utiliser ceux-ci pour générer notre code.

En guise d'exemple,pour bien comprendre le formalisme, voyons ce fichier XMI, qui représente un diagramme de classe avec une classe C1 qui possède deux attributs A1 et A2.

Exemple de fichier XMI

```
<?xml version="1.0" encoding="UTF-8"?>
  <xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
       xmlns:uml="http://schema.omg.org/spec/UML/2.1.1" xmi:version="2.1">
         <UML:Package xmi:id="ppp" xmi:label="p1">
             <ownedElement xmi:type="UML:Class" xmi:id="ccc" xmi:label="c1">
                     <feature xmi:type="UML:Attribute" xmi:label="a1"/>
                     <feature xmi:type="UML:Attribute" xmi:label="a2"/>
             </ownedElement>
         </UML:Package>
  </xmi:XMI>
```

FIG. 4.1 – Exemple de fichier XMI

La première ligne, comme dans tout fichier XML, est optionnelle. Elle désigne juste la version d'UML utilisée dans le fichier. Ensuite nous voyons, encore comme dans tout XML bien formé, que la seconde ligne est destinée à renseigner sur le namespace utilisé, pour savoir quels termes utiliser pour les noms de tags.
Les autres tags utilisent un attribut appelé " XMI : TYPE " qui renseigne sur le but du tag. Ainsi nous voyons un tag de type " Uml : class " qui désigne le classe C1 et deux tags de type " Uml : attribute " qui désignent bien entendu les attribut de la classe, à savoir a1 et a2.

Un fichier xmi représentant un diagramme d'états transitions se présente sous la même forme que celui-ci, mise à part le fait que les tags parleront de transitions, d'états, ... soit, d'éléments faisant partie d'un diagramme d'états transitions.

4.1.1 La conversion en xmi

Il existe beaucoup de logiciels sur le marché qui convertissent les diagrammes Uml en fichier xmi. (Visual Paradigm, Eclipse UML, ArgoUml, UModel, ...). Nous avons choisi le logiciel UModel, développé par Altova, pour des raisons de simplicité de l'interface graphique.
Il important de pouvoir disposer de ce fichier avant de commencer la génération du code car c'est à partir de celui-ci que nous allons aller rechercher les données nécessaires. Et ce grâce à un parsing de celui-ci. Ce parsing fera d'ailleurs l'objet de la section suivante du rapport de ce mémoire. [3]

4.2 Le parsing d'un fichier XMI

XMI[4] permet donc de définir un format d'échange selon les besoins de l'utilisateur. Il est donc essentiel pour le receveur d'un document XMI de pouvoir extraire les données du document. Cette opération est possible à l'aide d'un outil appelé **parseur**.

Le parseur permet de créer une structure hiérarchique contenant les données contenues dans le document XMI. On en distingue deux types selon l'approche qu'ils utilisent pour traiter le document :

- Les parseurs utilisant une approche hiérarchique : ceux-ci construisent une structure hiérarchique contenant des objets représentant les éléments du document, et dont les méthodes permettent d'accèder aux propriétés. La principale API utilisant cette approche est DOM (Document Object Model)
- Les parseurs basés sur un mode événementiel permettent de réagir à des événements(comme le début d'un élément, la fin d'un élément) et de renvoyer le résultat à l'application utilisant celui-ci. SAX (Simple API for XML) est la principale interface utilisant l'aspect événementiel.

Ainsi, on tend à associer l'approche hiérarchique avec DOM et l'approche événementielle avec SAX.

[3] Une version du fichier XMI représentant notre distributeur de bonbon sera disponible, pour le lecteur intéressé, en annexe de ce mémoire.

[4] Pour rappel XMI est l'acronyme de XML Metadata Interchange, ce qui veut dire qu'un fichier XMI est avant tout un fichier XML

Nous avons lors de ce travail utilisé l'approche hiérarchique des parseurs DOM car un document XMI représentant un diagramme UML se compose de plusieurs tags se faisant référence tout au long du fichier (cfr exemple de la classe avec attributs). Il n'est donc pas possible d'utiliser une approche événementielle car nous avons en permanence besoin de l'entièreté de l'arbre composant les tags du diagramme, or un parseur SAX ne garde en mémoire que ce qui est nécessaire à l'analyse de l'événement courant.

4.2.1 Fonctionnement du parseur

Avant de commencer l'explication, une petite remarque est nécessaire. En effet, nous rassemblons, dans ce mémoire, sous le vocable de " parseur " aussi bien l'outil capable d'analyser et d'extraire les données issues du fichier XMI que le programme informatique qui générera le code du système étudié. Ainsi, à la suite de l'exécution de ce parseur, le fichier XMI aura été analysé et le code associé au diagramme d'état transition aura été généré.

Dans un premier temps donc, il est nécessaire de construire une nouvelle instance de parseur DOM (ici au sens de la définition donnée plus haut) qui nous permettra de lire le fichier XMI. Cette étape ce fait au moyen du code ci-dessous :

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
document = builder.parse(new File("machineboisson.xml"));
```

FIG. 4.2 – Code parseur

Nous pouvons apercevoir dans la dernière ligne, la commande **builder.parse()** qui prend en paramètre le nom du fichier à analyser et qui permet dans lancer le parsing de celui-ci.

L'explication

Maintenant que nous avons à disposition tout les noeuds du fichier (pour rappel, le parseur DOM crée une structure hiérarchique avec TOUT les noeuds du fichier qu'il analyse), nous pouvons commencer à générer le code.
Cette étape se fera de manière séquentielle. C'est-à-dire que nous allons

générer toutes les classes d'un même type en même temps et ensuite nous continuerons la génération. Plus spécialement, dans un premier temps nous allons générer toutes les classes représentant un Etat, car il suffit dans ce cas d'aller rechercher les tags dont l'attribut " XMI :TYPE " (cfr section d'explication sur le XMI, plus avant) possède la valeur " XMI :TYPE = UML :State". Nous récupérons la valeur du nom de l'état et créons une classe JAVA qui portera ce nom. La même démarche sera effectuée à chaque fois qu'il sera nécessaire d'aller rechercher une information venant du fichier XMI et donc du diagramme.
Mais il existe des classes, dans la structure que nous avons mise au point, qui ne puisent pas toutes leurs informations dans le diagramme. Celles-ci seront toujours présentent quelque soit le contexte d'exécution. En effet nous parlons de la classe ETAT qui, mise à part le nom des méthodes de transitions aura toujours la même structure, de la classe contexte d'exécution et des classes permettant de mettre en place les états composite (cfr Fig. 3.4).
Ces types de classes seront généré automatiquement dès l'exécution du parseur afin de permettre de mettre en place la structure du diagramme de classe de la figure 3.4.

Chapitre 5

Les résultats

Nous allons dans ce dernier chapitre de ce rapport, analyser les résultats obtenus à l'aide de notre parseur/générateur. Mais avant de commencer, nous allons spécifier la marche à suivre pour utiliser celui-ci, afin d'être sur d'avoir bien compris son fonctionnement.
Tout part du diagramme d'états transitions du système que nous voulons étudier. Ensuite, il s'agit de transcrire ce diagramme en fichier XMI grâce à l'un des nombreux outils disponibles sur le marché (pour rappel nous avons utilisé UModel de Altova). Les étapes suivantes peuvent ce faire à l'aide de notre parseur. En effet, ayant le fichier XMI représentant le système à étudier, il suffit de lancer le programme en spécifiant bien le nom de ce fichier dans la ligne de commande adéquate (cfr section précédente) et les classes JAVA associées seront générées automatiquement et ce suivant la structure souhaitée par le pattern State.

5.1 L'analyse des résultats

Passons maintenant à l'analyse des résultats proprement dite, et voyons l'image ci-après.

Code java écrit à la main

```
/*
 * EtatVendu.java
 *
 * Created on 19 novembre 2007, 16:35
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

/**
 *
 * @author Pasquale
 */
public class EtatVendu extends Etat{
    /** Creates a new instance of EtatVendu */
    public EtatVendu(Distributeur distributeur) {
        super(distributeur);
        System.out.println("je crée etatvendu");

    }
    public void Delivrer(){
    getDistributeur().liberer();
    if(getDistributeur().getNombreBonbon()>0){
        getDistributeur().setEtat(getDistributeur().getEtatSansPiece());
    }
     else{
        System.out.println("Il n'y a plus de bonbon dans la machine");
        getDistributeur().setEtat(getDistributeur().getEtatEpuise());
    }
    };

}
```

Code JAVA généré automatiquement

```
public class  EtatVendu extends ETAT{

  public EtatVendu(ExecutionContext exctxt) {
    super(exctxt)
  }

  public void délivrerBonbon()  {
    if(bonbon=0)

        ExecutionContext.setEtat (Etat Epuisé);

    else if(Bonbon >0)

        ExecutionContext.setEtat (Etat SansPieces);
  }

}
```

FIG. 5.1 – Comparaison du code d'une classe écrit à la main et généré automatiquement

Comme nous le remarquons sur la figure, la partie de gauche représente le code que nous avons nous même écris à la main comme expliqué dans la section 3.3, et la partie de droite représente le code généré par notre parseur. Dans la partie écrite à la main, nous avons encadré les différents bouts de code qui devraient être issus des informations apportées par le diagramme d'états transitions, et nous pouvons les comparer au code généré pour nous rendre compte des ressemblances :

- Dans le diagramme UML, nous ne connaissons pas le nom du contexte d'exécution, c'est pourquoi nous lui avons donné le nom générique d'ExecutionContext.
- Nous pouvons voir aussi la méthode setEtat() qui est appelée par le contexte d'exécution et qui modifie l'état interne de la machine.
- Il y a aussi, bien sur, le constructeur qui a la même structure que celui écrit à la main.

Pour ne pas trop encombrer le rapport, nous n'avons pas comparé les autres classes d'états, car la comparaison serait du même ordre que celle effectuée à la page précédente.Mais le lecteur intéressé pourra accéder à celle-ci en annexe.

Un petit mot d'explication est toutefois nécessaire pour la classe Execution-Context.

```
public class  ExecutionContext {
 private Etat etat;
 public ExecutionContext() {

   Etat Etat Epuisé=new Etat Epuisé(this);
   Etat Etat SansPieces=new Etat SansPieces(this);
   Etat Etat Vendeur=new Etat Vendeur(this);
   Etat EtatAPiece=new EtatAPiece(this);
   Etat EtatVendu=new EtatVendu(this);
   Etat EtatGagnant=new EtatGagnant(this);
}
 public void tournéPoignée(){}
 public void délivrerBonbon(){}
 public void delivrer2bonbon(){}
 public void Remplir(){}
 public void insererPiece(){}
 public void ejecterPiece(){}
 public void setEtat(Etat etat){
   this.etat=etat;
 };
}
```

FIG. 5.2 – La classe ExecutionContext

En effet, comme nous le voyons, c'est cette classe qui instancie les états possibles du système étudié. Il est donc normal de retrouver ceux-ci dans le constructeur de cette classe. Cette manière d'opérer sera toujours la même quelque soit le contexte d'exécution.
De plus nous pouvons aussi apercevoir toutes les méthodes représentant les transitions dont l'implémentation finale est laissée aux soins de l'utilisateur qui connaît toutes les spécificités du système(comme nous avons pu le faire quand nous avons écrit nous même le code à la main).

Conclusions générales

Nous sommes arrivés maintenant au terme de ce mémoire. Celui-ci visait concrètement à implémenter un générateur de code automatique à partir des diagrammes d'états transitions UML. Nous avons pu démontrer que ceci est possible notamment grâce à l'aide du pattern State qui permet à un système de modifier son comportement suivant son changement d'état interne, ce qui convient parfaitement aux systèmes décrits par un diagramme d'états transitions. Cette génération passe toutefois par plusieurs étapes. En effet, nous avons d'abord transcrit le diagramme en un fichier XMI qui est un langage permettant de décrire les données présentent dans un diagramme UML de manière structurée (comme à l'image d'un ficher XML). Ensuite il nous a fallu écrire un parseur pour récupérer les données et pouvoir enfin les générer sous forme de classes JAVA construites suivant l'architecture proposée par le pattern State.

Le but premier du travail a été atteint. Toutefois, ce générateur est ouvert à de futurs travaux visant l'amélioration de celui-ci. En effet, nous pourrions par exemple, imaginer une sorte de programme intégrant un traducteur de langage UML vers XMI et le générateur lui-même. Celui-ci manque aussi peut-être d'une certaine ergonomie, problème qui pourrait être résolu par l'implémentation d'une interface graphique à partir de laquelle l'utilisateur pourrait directement générer le code.
D'un point de vue plus technique, ce générateur pourrait aussi, peut-être, être amélioré. En effet, son implémentation manque peut être d'une certaine souplesse. L'utilisation de celui-ci sur d'autres systèmes que notre distributeur pourrait faire l'objet d'une attention particulière lors d'un futur travail, afin de peut être mettre en évidence certaines lacunes de celui-ci.

Enfin, d'un point de vue plus personnel, je dois dire que toutes mes attentes ont été comblées lors de la réalisation de ce mémoire. En effet, la notion de programmation orientée objet ma beaucoup intéressé lors de mes études.

C'est pourquoi j'ai choisi ce mémoire qui ma permis d'étendre mes connaissances dans ce domaine, notamment en m'initiant à l'utilisation des design patterns qui sont très important lors du développement de logiciel.
De plus, j'ai pu aussi m'intéresser à d'autres sujets tels que la transcription de diagrammes en code grâce au XMI et le parsing de fichier XML.

Bibliographie

[1] Laurent Audiber. *UML 2.0.* Institut Universitaire de Technologie de Villetaneuse.

[2] Er.Freeman ; El.Freeman ; K.Sierra ; B.Bates. *Tête la première - Design Patterns.* O'Reilly, 2005.

[3] H. Bersini. *L'orienté objet.* Eyrolles, 2002.

[4] E.Zimanyi. *Analysis and Design.* 2002.

[5] fr.wikipedia.org/wiki/Patron de conception.

[6] Christophe Grosjean. Le xmi. *Décision Micro*, Avril 2001.

[7] Object Management Group. *MOF 2.0/XMI Mapping Version 2.1.1.* December 2007.

[8] Scolas Laurent. Conversion de modèles uml en format de graphes, 2007.

[9] John Lewis ; William Loftus. *Java Software Solutions.* Addison-Wesley, 2001.

[10] schema.omg.org/spec/XMI/2.1.

[11] J.Musset ; E.Juliot ; S.Lacrampe. *Acceleo références 2.1.* 2007.

[12] www.acceleo.org/pages/introduction/fr.

[13] www.commentcamarche.net/xml/xmldomsax.php3.

[14] www.dofactory.com/Patterns/PatternState.aspx.

[15] www.omg.org/technology/documents/formal/xmi.htm.

[16] www.smartstatestudio.com/xmi2sm/index.html.

Annexe

Le lecteur intéressé pourra trouver les codes sources du programme ainsi que les différents diagrammes utilisés pour ce programme au service IRIDIA de la faculté des sciences appliquées de l'Université Libre de Bruxelles.

Il pourra notamment y trouver :

- le code source du distributeur
- le diagramme de classe de celui-ci
- le diagramme d'états transitions
- le fichier XMI généré par Umodel et représentant le distributeur
- le code source du parseur XMI
- le code source des classes générées automatiquement par le parseur