Notation UML

Jacques Madelaine

17 novembre 2000

1 Introduction

UML signifie : « *The Unified Modeling Language For Object-Oriented Development* » ; on pourrait le traduire par *langage unifié pour la modélisation objet*, mais on pourra plus facilement en parler comme de la *notation* UML.

La notation UML [3] est aujourd'hui une norme de l'OMG (« *Object Management Group* ») qui est un consortium des principaux constructeurs de matériels et éditeurs de logiciels mondiaux. Il a été possible à cette notation unifiée de voir le jour car c'est un langage de modélisation objet et non pas une méthode objet. Bien sûr, UML doit beaucoup aux notations utilisées dans les méthodes Booch, OMT ou OOSE, mais elle peut servir de support pour d'autres méthodes.

UML définit les modèles suivants pour la représentation des systèmes :
- le modèle des classes qui décrit la structure statique ;
- le modèle des états qui exprime le comportement dynamique des objets ;
- le modèle des cas d'utilisation qui décrit les besoins de l'utilisateur ;
- le modèle d'interaction qui représente les scénarios et les flots de messages ;
- le modèle de réalisation qui montre les unités de travail ;
- le modèle de déploiement qui précise la répartition des processus.

Chacun de ces modèles est manipulé grâce à des diagrammes. Les modèles peuvent avoir plusieurs vues complètes ou partielles qui utilisent des diagrammes différents. UML définit neuf types de diagrammes différents :
- les diagrammes de classes ;
- les diagrammes d'objets ;
- les diagrammes de cas d'utilisation ;
- les diagrammes de séquences ;
- les diagrammes de collaboration ;
- les diagrammes d'états / transitions ;
- les diagrammes d'activités ;
- les diagrammes de composants ;
- les diagrammes de déploiement.

Le but d'UML est de pouvoir modéliser des systèmes entiers par des concepts objets et aussi de permettre d'échanger, entre les différents intervenants, les résultats des différentes étapes de modélisation. ces échanges seront possibles si la notation graphique est à la fois simple, intuitive, expressive et d'une sémantique précise.

Pour permettre la définition d'un modèle, on doit disposer d'un modèle de ce modèle ou *méta-modèle*. En fait le méta-modèle d'UML est UML lui-même. Ceci permet en même temps de définir proprement la notation et de prouver son expressivité. Il faut quand même noter que cela mène à des diagrammes dont, pour parler par euphémisme, le premier abord n'est pas facile.

On verra que l'on peut utiliser UML pour spécifier les diagrammes (figure 14).

2 Mécanismes communs

Nous avons comme mécanismes communs : stéréotypes, étiquettes, notes, contraintes, relation de dépendance, dichotomie type / instance et type / classe. Stéréotypes, étiquettes et contraintes permettent l'extension d'UML.

stéréotype Un élément spécialisé par un stéréotype permet de définir une classe supplémentaire du méta-modèle. Il est noté entre guillemets, ex. : « Utilitaire », « Énumération », « Métaclasse » pour des classes, « import » pour des relations de dépendance ;

étiquettes Paire (nom, valeur) ;

notes un simple commentaire ;

contraintes relations quelconques entre éléments de modélisation qui peuvent être exprimées en langage naturel, pseudo-code, expression logique, ... [1] ;

relation de dépendance relation d'utilisation unidirectionnelle entre éléments de modélisation (notes et contraintes sont en général des sources de relation de dépendance) ;

dichotomie type/instance et type/classe dans type/instance, le type dénote l'essence de l'élément et l'instance est une manifestation du type ;

[1] notons l'existence de R++, une extension de C++ qui permet de déclarer des contraintes sous forme de règles.

dans type/classe, la classe représentera la réalisation de la spécification énoncée dans le type.

3 Paquetages

Ils permettent de regrouper des éléments de modélisation. Sa représentation graphique se veut être un dossier : un rectangle avec un petit rectangle (onglet ?) en haut à gauche.

On utilise la même notation qu'en C++ à savoir :

`nom-de-paquetage : :nom-d-élément`

pour désigner un élément d'un paquetage à l'extérieur de ce dernier. On peut mettre des relations de dépendance (flèche en trait interrompu) entre paquetage utilisateur d'un élément défini dans un autre paquetage. Il est bien sûr déconseillé d'avoir des dépendances circulaires entre paquetages ce qui indique un couplage trop fort.

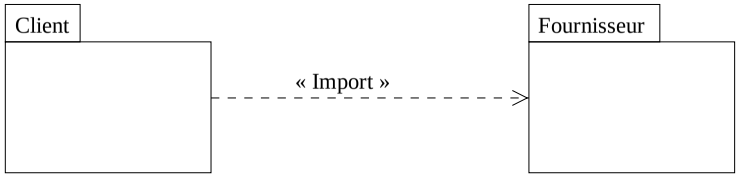

FIG. 1 – Deux paquetages.

4 Diagrammes de classes

Ces diagrammes permettent d'exprimer la structure statique d'un système.

4.1 Les classes

Une classe est représentée par un rectangle découpé en trois.

Le premier sert pour le nom de la classe, son éventuel stéréotype, et des étiquettes (<attribut> = <valeur>) permettant d'indiquer des informations générales. Le deuxième sert pour les attributs ou données membres . Le troisième pour les opérations ou fonctions membres.

Les stéréotypes de classes suivants sont définis par UML :

« `signal` » qui permet de déclencher une transaction dans un automate ;

« `interface` » qui ne déclare que des opérations (comme en Java) ;

« `métaclasse` » comme en Smalltalk ;

« `utilitaire` » qui ne regroupe que des fonctions statiques, comme un module C, ou comme la classe math de Java.

Les étiquettes peuvent permettre de préciser l'état de développement (État = testé), l'auteur (Auteur=Lui), ...

4.2 Membres : attributs et opérations

Les attributs et opérations peuvent être montrés de manière exhaustive ou non. Ils apparaissent comme une chaîne de caractères avec la syntaxe générale :
visibilité nomAttribut : *typeAttribut* = *valeurInitiale* { *propriété* }
visibilité nomOpération (*nomArgument* : *typeArgument* = *valeurParDéfaut*, ...) : *typeRetourné* { *propriété* }

UML définit trois niveaux de visibilité qui sont les mêmes qu'en C++ : private noté `-`, public noté `+` et protected noté `#`.

L'indication de propriétés est optionnelle.

Le nom d'une opération abstraite (ou virtuelle pure) apparaît en *italique*. Les membres attributs ou opérations de classe (membres statiques en C++) sont indiqués en souligné[2].

4.3 Classes paramétrables

Les classes paramétrables correspondent aux *template* C++ ou aux classes génériques d'Eiffel. Les paramètres de la classe sont écrits dans un rectangle pointillé en haut à droite du rectangle de la classe. L'instanciation d'une classe paramétrable est notée comme en C++ avec les paramètres entre < et >. Une relation de dépendance vers la classe paramétrable est aussi indiquée.

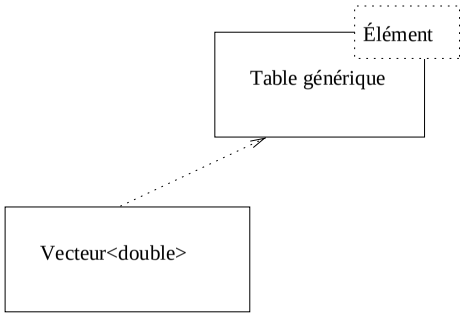

FIG. 2 – Classe paramétrable instanciée.

[2]On peut en effet considérer, pour un attribut, que c'est un objet visible dans la classe et qu'il est noté comme tel (voir 5).

4.4 Associations

Les associations représentent des relations structurelles entre classes d'objets. Elles sont représentées par une ligne reliant les deux classes, ou par l'intermédiaire d'un petit losange relié par des lignes s'il y a plusieurs classes impliquées. Les associations peuvent être nommées, auquel cas leur nom apparaît en italique au milieu de la ligne ; un signe ▶ (ou plus simplement >) peut préciser le sens de lecture. Une association peut très bien avoir plusieurs attributs en propre ; ils sont alors représentés par dans un rectangle de classe relié à l'association par un trait interrompu fin codant une représentation de dépendance. Bien sûr on peut également nommer les rôles des associations. Les figures 3 à 8 représentent différents exemples.

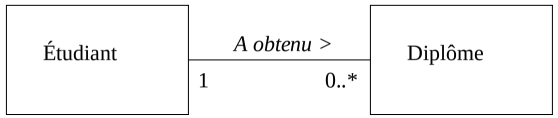

Fig. 3 – Exemple d'association simple.

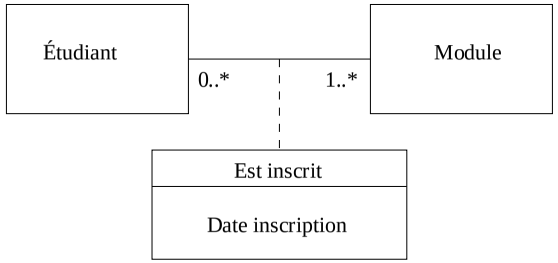

Fig. 4 – Exemple d'association possédant un attribut.

L'exemple de la figure 9 peut nous permettre de faire varier les cardinalités. Il montre les cardinalités sous le régime de la monogamie. La polygamie s'exprimera par des cardinalités 0..1 du côté Homme et 0..n ou 0..* du côté Femme. La polyandrie avec mariage obligatoire pour les hommes s'exprimera avec des cardinalités 0..* du côté Homme et 1 du côté Femme.

Avec des associations n-aire entre trois classes ou plus, la cardinalité sur un rôle représente le nombre potentiel d'instances de l'association qui peuvent exister quand les autres $n-1$ valeurs sont fixées.

On peut aussi *qualifier* une association en indiquant un ou plusieurs attributs, dans un petit rectangle, dont les valeurs utilisées avec une instance

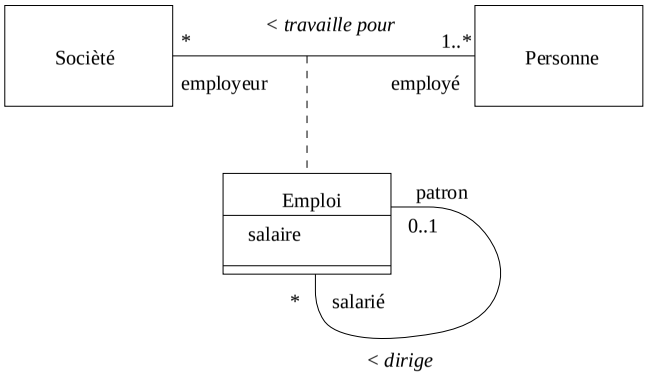

Fig. 5 – Exemple d'association étant elle-même source d'association.

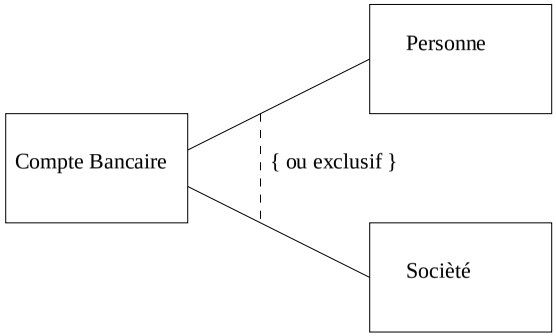

Fig. 6 – Exemple d'association avec contrainte (que l'on pourra aussi représenter avec un héritage, voir figure 16).

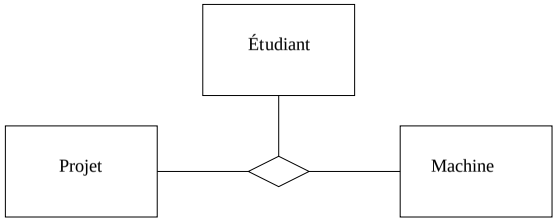

Fig. 7 – Exemple d'association ternaire.

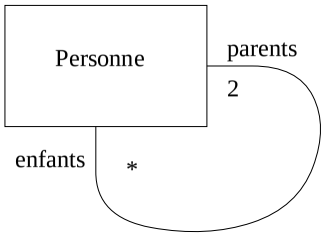

FIG. 8 – Exemple d'association où les rôles sont essentiels.

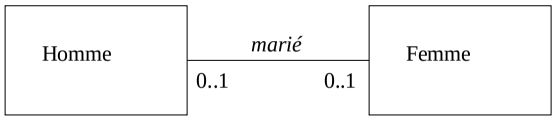

FIG. 9 – Exemple d'association pour comprendre les cardinalités.

sélectionneront 0, 1 ou plusieurs liens. Ces attributs font parties de l'association ; cette notation est en fait équivalente à celle qui représente une classe dépendant de l'association comme dans la figure 4. Ils peuvent ainsi servir dans des expressions de navigation qui permettent d'exprimer des contraintes.

4.5 Agrégation

Une agrégation est une association non symétrique ; elle est représentée par un petit losange du côté de l'agrégat. Les cardinalités peuvent être supérieures à 1, comme dans l'exemple des copropriétaires de la figure 10. On peut également indiqué des contraintes écrite entre accolades. Le losange vide indique une agrégation faiblement couplée. Un losange plein indiquera une agrégation forte ou composition.

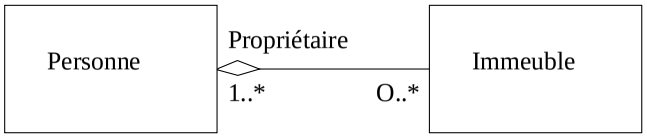

FIG. 10 – Exemple de représentation d'agrégation.

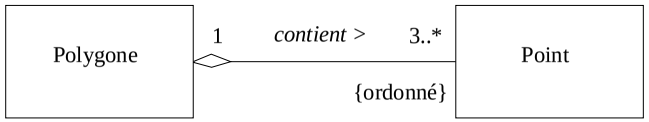

FIG. 11 – Agrégation avec contrainte.

4.6 Composition

La composition représente une agrégation avec une forte dépendance composant / composé et une coïncidence des durées de vies des parties et du tout. La multiplicité de l'agrégat ne peut pas dépasser 1 (pas de partage) et ne peut pas changer (les liens sont fixes). Plusieurs notations sont proposées en UML voir figure 12.

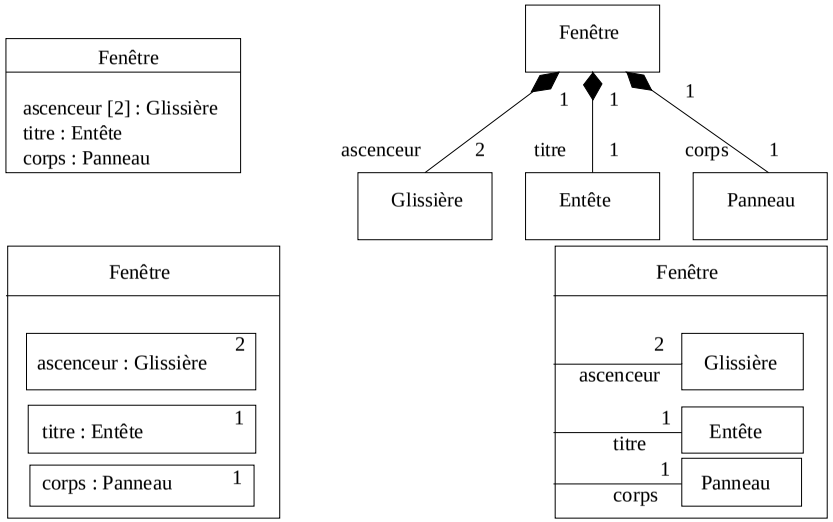

FIG. 12 – Exemple de représentation de composition.

4.7 Éléments dérivés

Certains éléments comme des attributs ou associations peuvent être dérivable. Les éléments dérivés sont notés avec un / liminaire. Une contrainte (notée entre accolades) permet de définir la dérivation. Voir figure 13.

Il existe aussi dans UML un petit langage de navigation qui permet d'exprimer

des contraintes. La notation pointée permet de choisir un élément en suivant le rôle d'un lien, les crochets permettent de dénoter un ensemble restreint par une condition. Voici deux exemples :

vol.pilote.nombreHeureVol > vol.avion.nombreHeureMini

societé.employé[fonction="Directeur" and count(employé) > 10]

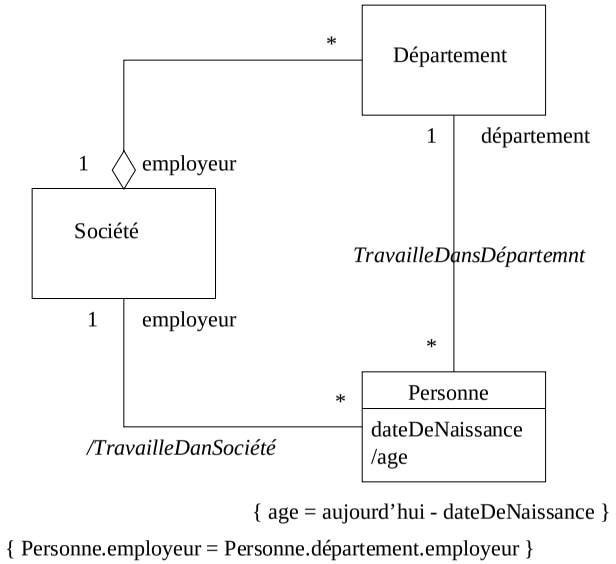

Fig. 13 – Exemple d'éléments dérivés.

4.8 Généralisation

Les généralisations d'UML permettent d'exprimer aussi bien l'héritage simple ou multiple que la classification simple ou dynamique. Elles sont notées avec un triangle dont la pointe est orientée vers la classe la plus générale ou la classe de base. Les classifications peuvent être précisées par des contraintes notées entre accolades :

complet qui spécifie que tous les sous-types ont étés spécifiés et que l'on ne peut pas en ajouter ;

incomplet qui est la contrainte par défaut, qui autorise à ajouter des sous-types ; on peut même préciser sur le diagramme une ellipse à la place d'une classe pour bien le montrer ;

disjoint ou exclusif qui est également la contrainte par défaut, qui précise que des instances ne pourront être instance que d'un seul de ses sous-types (i.e. pas d'héritage multiple possible) ;

chevauchement ou inclusif qui précise que des instances peuvent avoir plus d'un des sous-types comme type.

Le premier exemple est tiré de la définition de la sémantique de diagrammes UML voir figure 14. La figure 15 donne un exemple d'utilisation des contraintes pour spécifier un héritage multiple.

On peut aussi utiliser l'héritage pour spécifier des contraintes dans une association comme sur la figure 16.

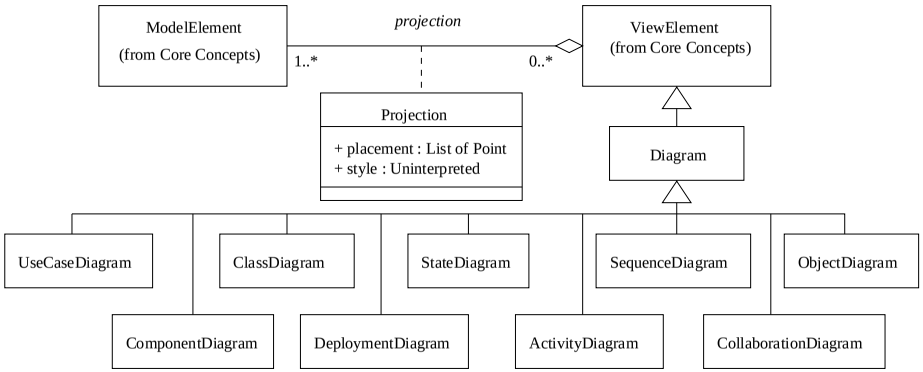

Fig. 14 – Définition des diagrammes UML en UML.

4.9 Classe abstraite

Une classe abstraite est notée comme une classe mais avec un nom écrit en *italique* comme pour une opération abstraite.

4.10 Interface

On peut représenter une interface de deux façons, soit avec un petit cercle relier à une classe la définissant, soit avec le stéréotype « interface ».

4.11 Dépendances entre classes

Des relations de dépendances entre classes peuvent être notées par une flèche en trait interrompu. Les stéréotypes « amie », « instancie », « appelle » peuvent être utilisés.

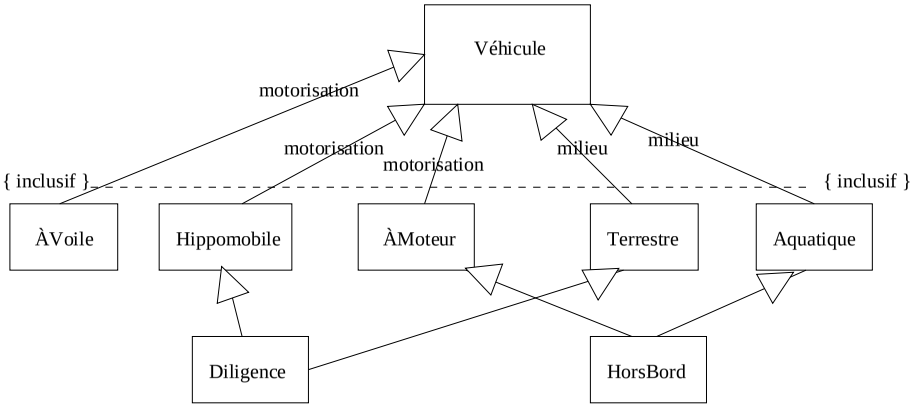

Fig. 15 – Exemple d'héritage multiple.

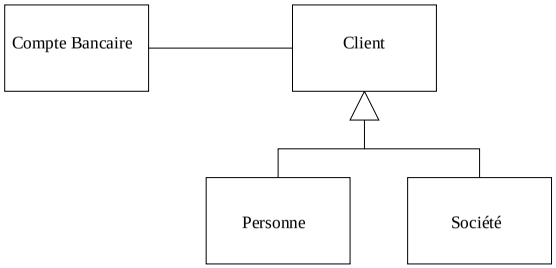

Fig. 16 – Exemple d'association reprenant la figure 6.

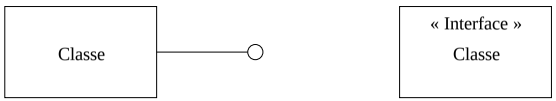

Fig. 17 – Représentation d'interfaces.

5 Diagrammes d'objets

Un objet se représente comme une classe à l'exception de son nom qui est souligné (voir figure 18). Les instances de relations sont des liens entre objets (voir figure 19). On peut préciser le nom de la classe de l'objet et le préfixer au besoin par les noms de paquetages. On peut même rappeler le stéréotype de la classe.

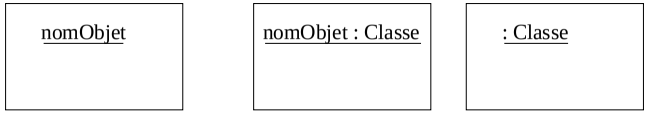

Fig. 18 – Exemple d'objets ou instances de classes.

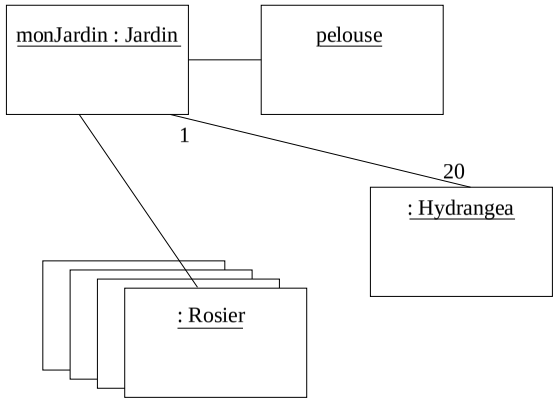

Fig. 19 – Exemple de liens entre objets.

6 Cas d'utilisation

Les cas d'utilisation (*use cases*) ont été formalisés par Ivar Jacobson. Ils décrivent sous forme d'actions et de réactions, le comportement d'un système du point de vue d'un utilisateur. Ils permettent de décrire les interactions système ou sous-système et environnement. Ces diagrammes étaient absents des méthodes Booch ou OMT et vont modéliser les différentes vues que les utilisateurs ont du système ou leurs besoins éventuellement contradictoires ! Ces diagrammes mélangent des schémas à la signification intuitive évidente avec des

descriptions dans la langue des utilisateurs des préoccupations premières des futurs utilisateurs.

Le modèle des cas d'utilisation comprend les acteurs, le système et les cas d'utilisation eux-mêmes. Les acteurs sont représentés par un homme très stylisé. Leur utilisation du système est représentée par une relation de dépendance avec un cas d'utilisation. Bien sûr, un même utilisateur réel peut tour à tour jouer le rôle de plusieurs acteurs. Les cas d'utilisation représentés par des ellipses sont des classes dont les instances sont les scénarios ; chaque scénario correspond au flot de messages échangés par les objets au cours de l'utilisation particulière.

Les relations entre cas d'utilisation sont les relations de communication d'un acteur avec un cas, les relations d'utilisation d'un cas par un autre (stéréotype « Utilise ») et les relations d'extension de deux cas (stéréotype « Étend »). La figure 20 montre un exemple de commandes de produits directement dans le magasin ou par l'intermédiaire du Web.

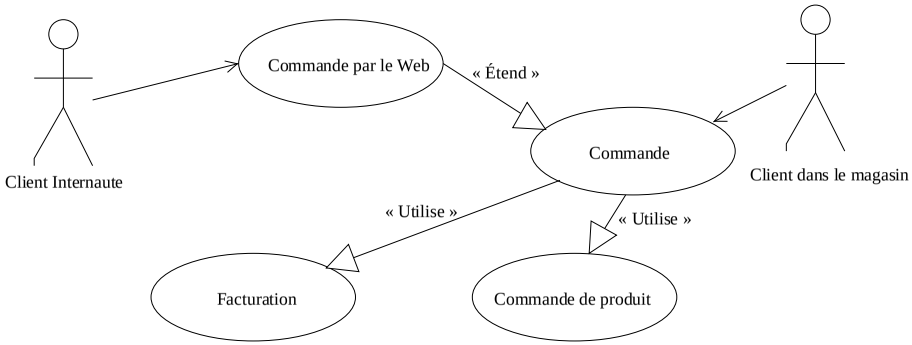

FIG. 20 – Cas d'utilisations pour un magasin avec un service Web.

Les cas d'utilisation présentés ici sont ceux de la version V1.1. Avec la version V1.3, ils ont légèrement changés et l'exemple précédent se représentera comme sur la figure 21.

7 Diagrammes de collaboration

Ils servent à montrer des interactions entre objets sous forme de message envoyés. Ils sont numérotés pour indiquer une séquence. Un acteur externe peut même être représenté. Voir figure 22.

Il est également possible de représenter des structures de contrôle : alternatives ou boucles. La condition d'une alternative est écrite entre crochets ([]). Une boucle peut être précisée par exemple par : `*[i :=1..n] : message(args)`. Le

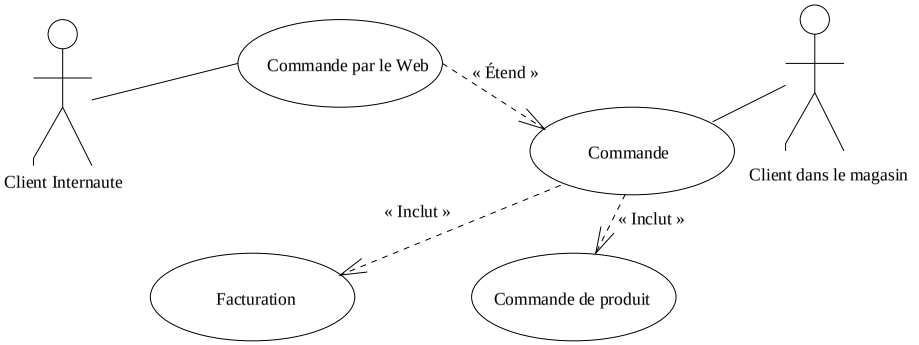

FIG. 21 – Cas d'utilisations pour un magasin avec un service Web. Notation V1.3.

résultat d'un message qui pourra servir comme argument d'un autre message est définit avec : `res := message(args)`.

On peut aussi simplifier un diagramme de collaboration en utilisant des *design patterns* [1], comme dans la figure 23.

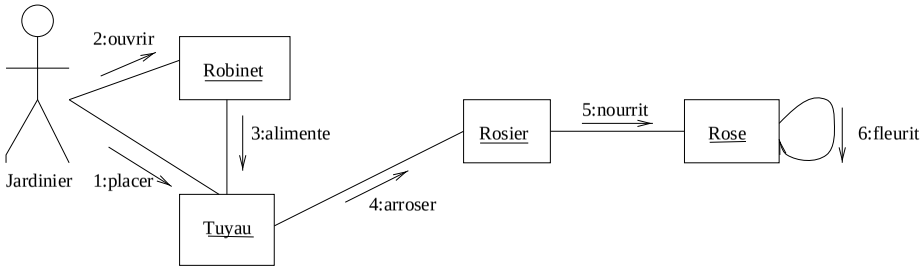

FIG. 22 – Exemple de collaboration.

8 Diagramme de séquences

Les diagrammes de séquences sont, comme les diagrammes de collaboration, des diagrammes permettant de montrer le comportement d'objets dans un scénario, mais représentent plus directement l'aspect chronologique des interactions.

Chaque objet est matérialisé comme dans un diagramme d'objet et possède une ligne verticale en trait interrompu qui représente la ligne de vie de l'objet.

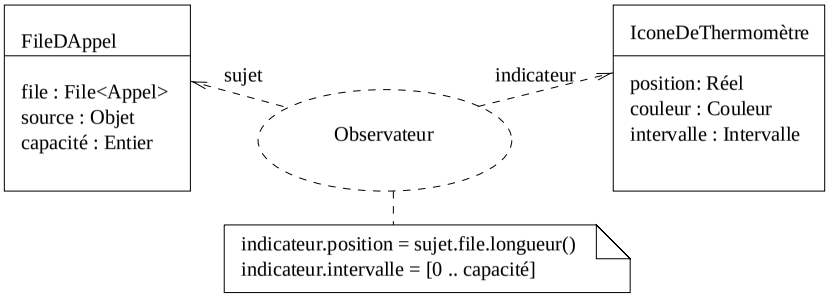

FIG. 23 – Exemple de *design pattern*.

La figure 24 montre les différents types de messages que l'on peut représenter, la figure 25représente les activations des objets ce qui permet d'avoir des réponses de messages synchrones implicites (ce qui modélise parfaitement les envois de messages du C++). La figure 26 présente une étude de cas : un distributeur de boissons.

On peut imposer des contraintes de temps (écrites entre accolades) en nommant des instants correspondants aux envois ou réceptions de messages. Comme dans les diagrammes de collaboration, on peut préciser des envois multiples, par exemple :

`*[i :=1..nbFenetres] Raffraichir(i)`,
ou `*[nonFini]Travailler`,
ou bien de simple condition`[piece.nonFausse()]Continuer`.
Pour des situations plus complexes, on peut avantageusement utiliser du pseudo-code.

9 Diagramme d'états / transitions

Les diagrammes d'états / transitions s'inspirent des *statecharts*. Si les diagrammes de classes sont des abstractions de la structure statique des objets, les diagrammes d'états / transitions sont des abstractions des comportements possibles. Chaque objet suivra le comportement décrit dans l'automate associé à sa classe. On peut aussi associer un automate à un objet composite ou à un cas d'utilisation.

Les états sont représentés dans des rectangles aux coins arrondis. Les transitions sont représentées par des flèches. Comme dans un réseau de Petri, les transitions sont franchies instantanément. Les évènements déterminent quels sont les transitions qui doivent être franchies. Un événement est défini par : son nom, une liste de paramètres, l'objet expéditeur, l'objet destinataire, la description de la signification de l'événement. Chaque transition est étiquetée par une chaîne de format général suivant :

événement(args) [*garde*] / *action* ^ *cible.événement(args)*

L'événement permet de déclencher la transition, la garde indique une condition qui permet de choisir entre plusieurs transitions ; le franchissement de la transition va provoquer l'exécution de l'action, puis éventuellement d'envoyer un événement sur une cible. Un exemple général est :

bouton3-enfoncé(position)[fenetre.estIntérieur(position)]
/objet := quelObjet(position) ^ objet.inverseVideo()

Un état peut être raffiné en sous-états ; chacun de ses sous-états pouvant être récursivement raffiné en de nouveaux sous-états. Un état peut correspondre à un seul des sous-états (sous-système séquentiel) ou à plusieurs sous-états (sous-système parallèle).

La figure 27 donne un exemple d'automate dont l'état actif est développé dans la figure 28.

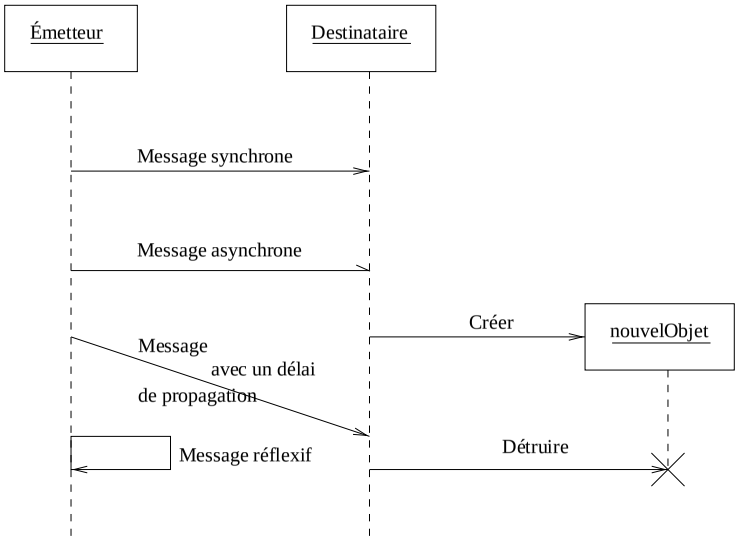

FIG. 24 – Exemple de messages.

10 Diagrammes d'activités

Les diagrammes d'activités sont des variantes de diagrammes d'états / transitions adaptés à la représentation du déroulement interne d'une opération ou

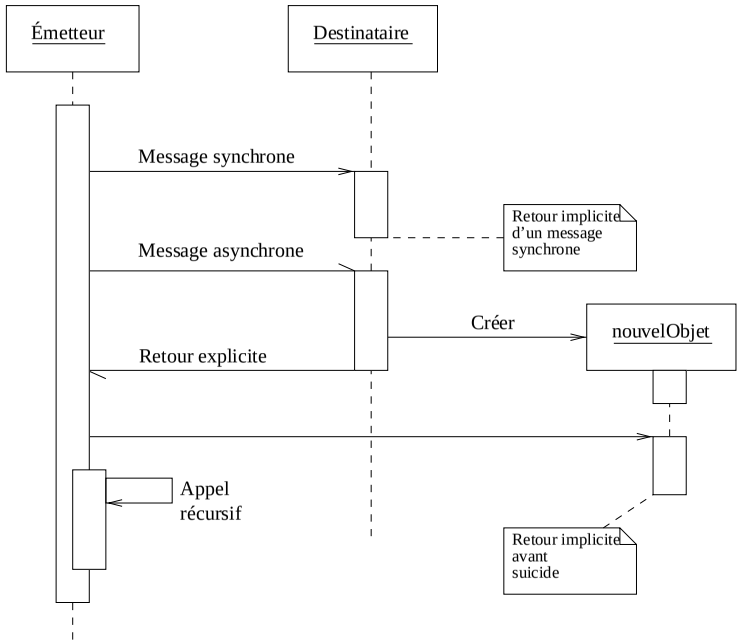

FIG. 25 – Exemple de séquences avec représentation d'activation.

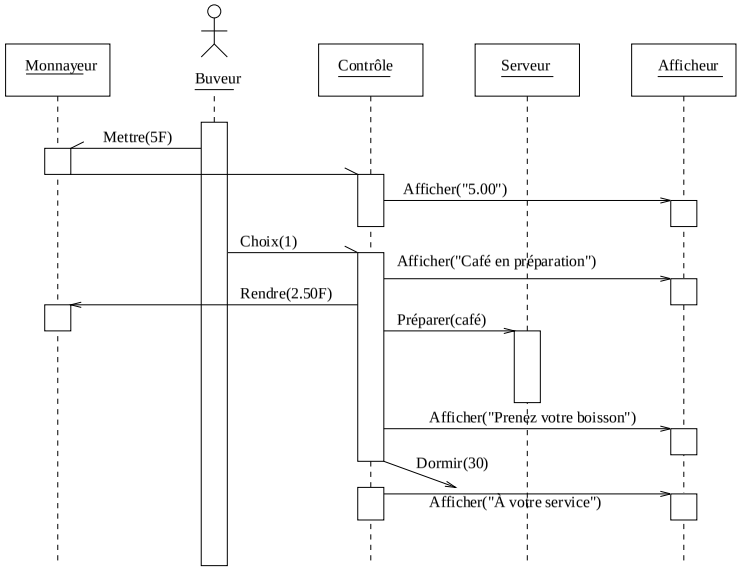

FIG. 26 – Un distributeur automatique de boisson.

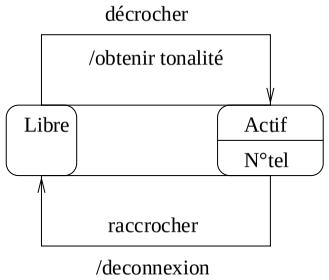

FIG. 27 – Comportement du téléphone.

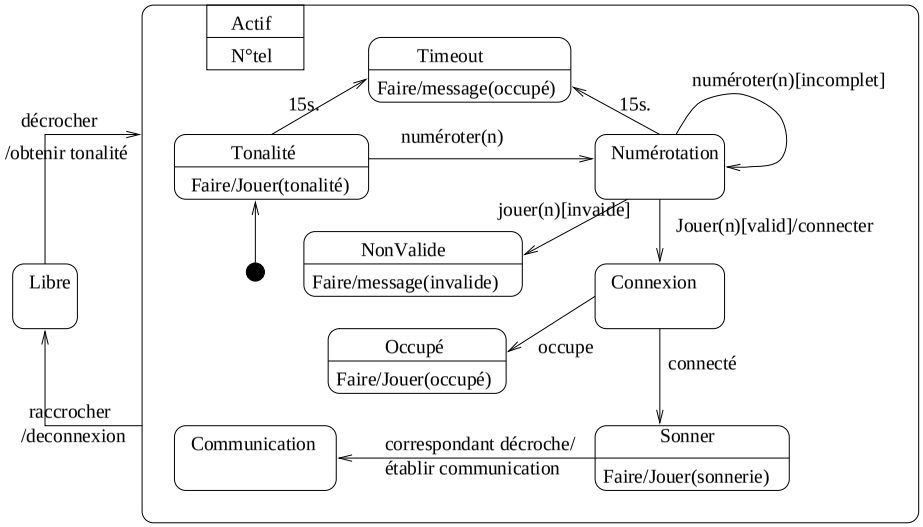

Fig. 28 – Comportement développé du téléphone.

d'un cas d'utilisation. C'est une vision simplifiée de l'enchaînement d'activités. Ce diagramme permet aussi de mêler les flots d'action et d'objets, voir figure 30.

11 Diagrammes de composants

Les diagrammes de composants montrent les choix de réalisation en décrivant les modules (figure 31), des programmes et sous-programmes (figure 32), des tâches ou *threads* (figure 33) et leurs dépendances.

12 Diagrammes de déploiement

Les diagrammes de déploiement montrent les dispositifs matériels mis en œuvre dans le système. Chaque élément physique est représenté dans un cube, on précise la nature de l'équipement par un stéréotype.

13 Glossaire

omt *Object Modelling Technique* de James Rumbauch ;

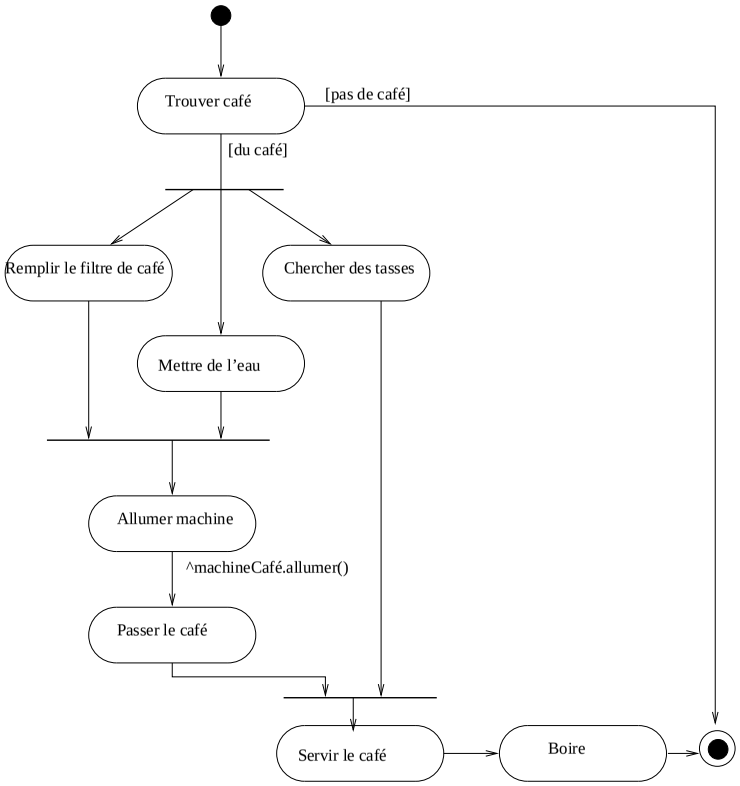

Fig. 29 – Comment faire du café ?

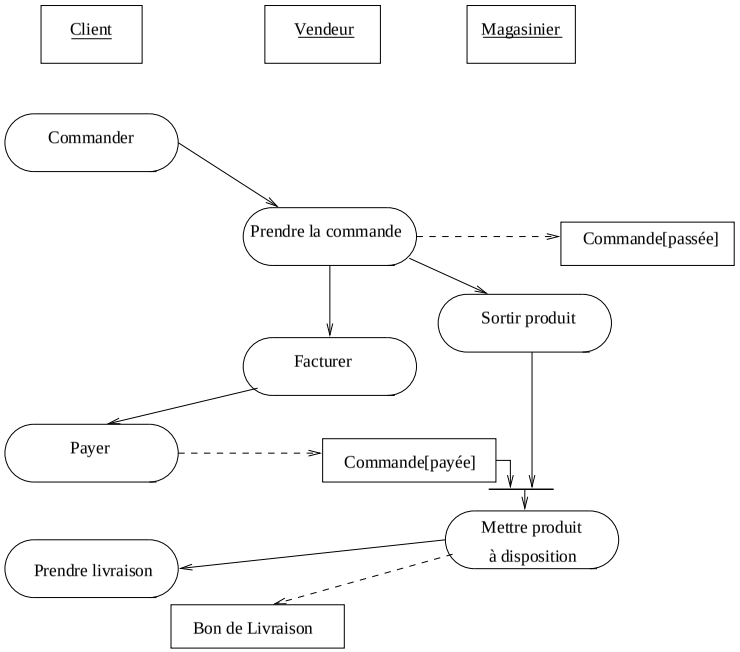

FIG. 30 – Acheter de l'electro-ménager.

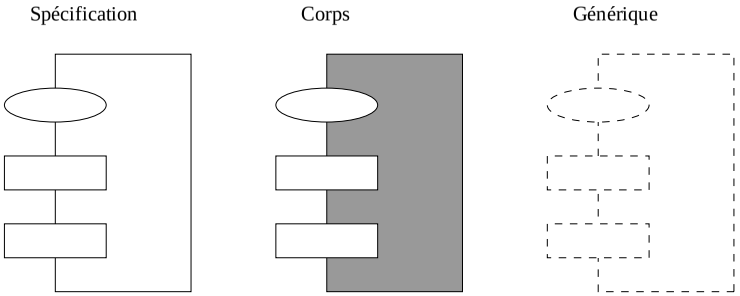

FIG. 31 – Représentation de modules.

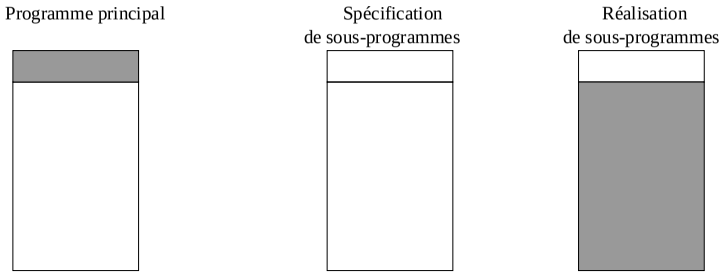

FIG. 32 – Représentation de programmes et sous-programmes.

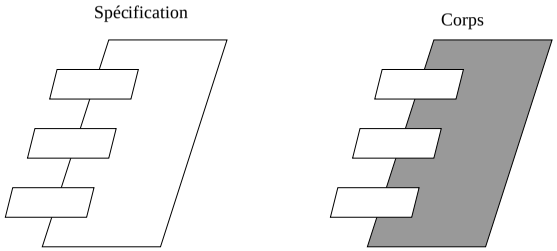

FIG. 33 – Représentation de tâches.

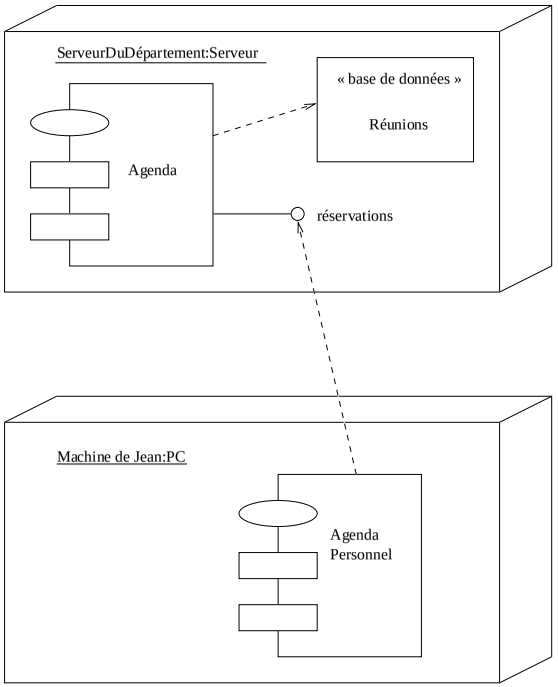

FIG. 34 – Un diagramme de déploiement.

uml *The Unified Modelling Language For Object-Oriented Development*, norme de l'OMG ;

omg *Object Management Group*, consortium des principaux constructeurs de matériels et éditeurs de logiciels mondiaux (`http ://www.omg.org`)

oose *Object Oriented Software Engineering* ou *Objectory*

Références

[1] E. Gamma, Helm R., Jonhson R., and Vlissides J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Pierre-Alain Muller. *Modélisation objet avec UML*. 1997.

[3] Rational Software. Unified modelling language. Technical report, Rational Software Corporation, 1997.