UNIFIED MODELING LANGUAGE (UML)

1. Introduction

UML (*Unified Modeling Language*) est une méthode de modélisation orientée objet développée en réponse à l'appel à propositions lancé par l'OMG (*Object Management Group*) dans le but de définir la notation standard pour la modélisation des applications construites à l'aide d'objets. Elle est héritée de plusieurs autres méthodes telles que OMT (*Object Modeling Technique*) et OOSE (*Object Oriented Software Engineering*) et Booch. Les principaux auteurs de la notation UML sont Grady Booch, Ivar Jacobson et Jim Rumbaugh.

Elle est utilisée pour spécifier un logiciel et/ou pour concevoir un logiciel. Dans la spécification, le modèle décrit les classes et les cas d'utilisation vus de l'utilisateur final du logiciel. Le modèle produit par une conception orientée objet est en général une extension du modèle issu de la spécification. Il enrichit ce dernier de classes, dites techniques, qui n'intéressent pas l'utilisateur final du logiciel mais seulement ses concepteurs. Il comprend les modèles des classes, des états et d'interaction. UML est également utilisée dans les phases terminales du développement avec les modèles de réalisation et de déploiement.

UML est une méthode utilisant une représentation graphique. L'usage d'une représentation graphique est un complément excellent à celui de représentions textuelles. En effet, l'une comme l'autre sont ambigues mais leur utilisation simultanée permet de diminuer les ambiguités de chacune d'elle. Un dessin permet bien souvent d'exprimer clairement ce qu'un texte exprime difficilement et un bon commentaire permet d'enrichir une figure.

Il est nécessaire de préciser qu'une méthode telle que UML ne suffit pas à produire un développement de logiciel de qualité à elle seule. En effet, UML n'est qu'un formalisme, ou plutôt un ensemble de formalismes permettant d'appréhender un problème ou un domaine et de le modéliser, ni plus ni moins. Un formalisme n'est qu'un outil. Le succès du développement du logiciel dépend évidemment de la bonne utilisation d'une méthode comme UML mais il dépend surtout de la façon dont on utilise cette méthode à l'intérieur du cycle de développement du logiciel.

Dans UML, il existe plusieurs formalismes ou « modèles » :

- le modèle *des classes*
- le modèle *des états*
- le modèle *des cas d'utilisation*
- le modèle *d'interaction*
- le modèle *de réalisation*
- le modèle *de déploiement*

Le modèle des classes est le plus utile. C'est un formalisme pour représenter les concepts usuels de l'orienté objet. Le modèle des états et le modèle d'interaction permettent de représenter la dynamique des objets. Le modèle des cas d'utilisation permet de décrire les besoins de l'utilisateur final du logiciel. Le modèle de réalisation et le modèle de déploiement, moins importants que les autres modèles de UML, ne seront pas décrits par ce document.

Références :

Pierre-Alain Muller – Modélisation objet avec UML, Eyrolles 1997.
Rumbaugh – OMT, cours et exercices, Eyrolles.

Ce document ne présente que les modèles traitant de la technologie objet. Il présente le modèle des classes dans la partie 2, les modèles des états et d'interaction dans la partie 3. Le modèle des cas d'utilisation est décrit dans la partie 4. Enfin, la partie 5 effectue un lien possible entre une conception UML et une programmation Java ou C++.

2. Le modèle des classes

Le *modèle des classes* d'UML saisit la structure statique d'un système en montrant les objets dans le système, les relations entre les objets, les attributs et les opérations qui caractérisent chaque classe d'objets. C'est le plus important des modèles d'UML. Il utilise plusieurs types de diagrammes :

les *diagrammes de classes*,
les *diagrammes d'objets*.

2.1. Objets et classes

Objets

Un *objet* est une entité qui a un *sens* dans le contexte de l'application.

Un objet possède une *identité*.

Instance d'objet -> référence à une chose précise
Classe d'objets -> référence à un groupe de choses similaires

Classes

Une *classe* d'objets décrit un groupe d'objet ayant des propriétés similaires, un comportement commun, des relations communes avec les autres objets.

« Personne », « Société », « Animal », « Fenêtre » sont des classes d'objets.

Diagrammes de classes et diagrammes d'objets

Les *diagrammes de classes* permettent de modéliser les classes. Les *diagrammes d'objets* permettent de modéliser les *instances*. En UML, le mot « objet » est souvent lié à la notion d'instance alors qu'en orienté objet usuel, le mot « objet » est souvent lié aux deux notions de classe et d'instance. Dans ce document, nous restons le plus explicite possible en gardant le mot instance.

exemple d'une classe et d'une instance (objet) :

« Personne » est une classe. « Candice » est une instance qui appartient à la classe « Personne ». « Bob » est une instance dont la classe n'est pas précisée. La troisième instance est un objet *anonyme* dont la classe est précisée mais pas le nom.

Les instances et classes sont représentées par des rectangles.
Les noms des instances et des classes commencent par une majuscule.
Un nom de classe est toujours au singulier : pas de S à la fin, même si conceptuellement une classe est un ensemble d'instances.
Le nom d'une instance est suivi de **:** et du nom de la classe à laquelle elle appartient. Le tout est souligné.

Un diagramme de classes ne peut pas contenir plusieurs fois la même classe. Un rectangle unique correspond à chaque classe du diagramme.

Commentaires

Un diagramme UML peut être précisé à l'aide commentaires. Un commentaire est écrit entre accolades.

{Ceci est un commentaire}

Les commentaires sont utiles pour minimiser les ambiguités des diagrammes.

Attributs

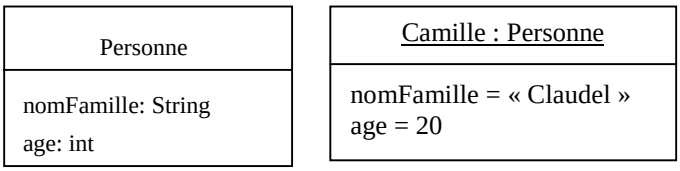
Un *attribut* est une propriété commune à tous les objets d'une classe. Par exemple, « nomFamille » et « âge » peuvent être des attributs de la classe « Personne ».

Pour chaque instance d'une classe ayant un attribut, l'instance possède cet attribut et cet attribut peut prendre une *valeur* pour une instance d'un objet.

Un attribut doit contenir une valeur pure et pas un autre objet. Sinon on préfère utiliser la notion de relation (cf. plus loin).

Les attributs sont définis dans la 2ème partie du rectangle désignant la classe. On fait suivre le nom de chaque attribut par **:** et le type de l'attribut.

Pour décrire les attributs d'une instance, on utilise également un deuxième rectangle contenant la liste des attributs avec leur valeur. Par exemple, le diagramme ci-dessous montre la classe « Personne » avec les attributs « nomFamille » et « age » et l'instance « Camille » de la classe « Personne » dont le nom de famille est « Claudel » et l'age est 20 ans.



Opérations et méthodes

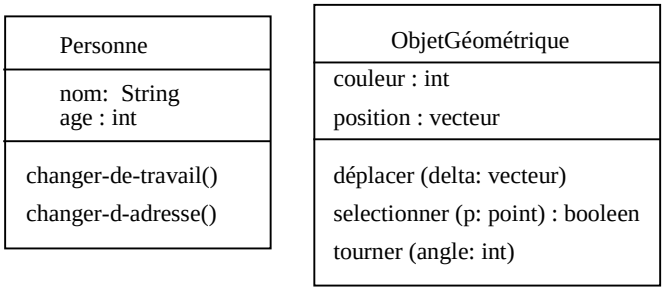
Une *opération* est associée à une classe ; c'est une fonction ou une transformation qui peut être appliquée aux objets d'une classe. Par exemple, « Embaucher », « licencier », « payer » sont des opérations de la classe « Société ». Chaque opération possède un objet cible ou instance sur lequel elle s'applique.

La même opération peut s'appliquer sur plusieurs classes d'objets : on dit qu'elle est *polymorphe*. Une *méthode* est l'implémentation d'une opération sur une classe. En programmation orientée objet (Java, C++) on parle de méthodes et en spécification et conception orientées objet (OMT, UML), on parle d'opérations.

Une opération peut avoir des *paramètres*. (de la même manière qu'une fonction ou procédure en possède en programmation classique).

Quand une opération possède plusieurs méthodes dans plusieurs classes, elles doivent avoir la même signature.

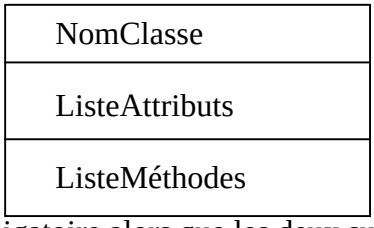
signature : nombre et types des arguments. *propriété* est un terme générique pour désigner un attribut ou une opération.



Sur cet exemple, la classe « Personne » possède deux opérations ou méthodes : « changer-de-travail() » et « changer-d-adresse() » sans paramètre ni valeur de retour. La classe « ObjetGéométrique » possède 3 opérations ou méthodes : la méthode « déplacer() » possédant le paramètre « delta » de type « vecteur », la méthode « sélectionner() » possédant le paramètre « p » de type « point » et retournant une valeur de type « booleen », et la méthode « tourner() » sans paramètre ni valeur de retour.

Une *requête* est une opération qui se contente de calculer une valeur fonctionnelle sans modifier un objet. (par opposition aux opérations qui modifient un ou plusieurs objets). Les *attributs de base* caractérisent l'objet. Les *attributs dérivés* se calculent à partir des attributs de base.

En résumé, une classe est représentée par un rectangle possèdant jusqu'à 3 zones :



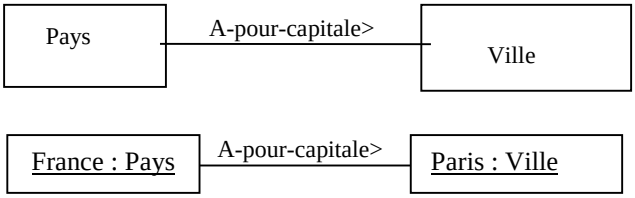
La première zone est obligatoire alors que les deux suivantes sont optionnelles.

2.2. Liens et associations

Les *liens* permettent d'établir des relations entre objets (ou instances). Les *associations* permettent d'établir des relations entre classes. Un *lien* est une connexion entre des instances d'objets. Une *association* décrit un groupe de liens ayant un sens commun et c'est une connexion entre des classes d'objets.

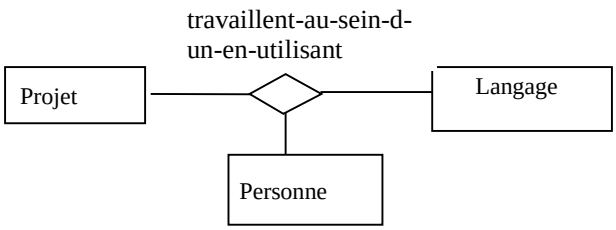
« travaille-pour » est une association de la classe « Personne » avec la classe « Société ».
« a-pour-capitale » est une association de la classe « Pays » avec la classe « Ville ».

Pour une association *binaire* : il existe un *rôle vers l'avant* et un *rôle inverse*. UML permet de spécifier le sens de lecture de l'association avec les symboles > et <. « emploie » est le rôle inverse de « travaille-pour ». « est-la-capitale-de » est le rôle inverse de « a-pour-capitale »



Une association est souvent implémentée sous forme de *pointeurs* dans les langages de programmation. En UML ou tout autre méthode de modélisation, cette utilisation de pointeurs est interdite et remplacée par la notion d'association. Plus loin, nous verrons comment passer d'une association modélisée avec une méthode orientée objet (UML, OMT) à des attributs pointeurs d'un langage orienté objet (Java, C++).

Une association peut relier une, deux ou plusieurs classes. Dans le cas où elle relie n classes elle est dite *n-aire* et dans le cas où elle relie deux classes, elle est dite *binaire*. Le symbole représentant une association n-aire est le losange. Il est placé au centre de l'association ; les n branches de l'association l'ont pour origine et ont les n classes de l'association pour destination.



Ci-dessus, la figure représente une association *ternaire* exprimant l'idée que des « personnes » travaillent au sein d'un « projet » en utilisant un « langage ».

L'utilisation des associations n-aires est souvent très ambigüe mais elle sert pour esquisser un modèle lorsque la précision est inutile, par exemple au début d'une spécifications de besoins. Pour être plus précis sur le diagramme ci-dessus, il faut remplacer l'association par trois associations binaires reliant les 3 classes deux à deux.

Multiplicité

Pour les deux rôles d'une association binaire, la *multiplicité* est un ensemble de valeurs indiquant le nombre possible d'instances de la classe destination du rôle qui peuvent être reliées à une instance de la classe origine du rôle. En première approximation, on la décrit souvent comme étant « un » (l'intervalle [1,1]) ou « plusieurs » (l'intervalle [0,+∞[). Il existe donc des associations « un-à-un » ou « un-a-plusieurs » ou « plusieurs-à-plusieurs ».
La multiplicité est théoriquement un ensemble de valeurs mais en pratique c'est souvent un intervalle.

Par exemple, l'association « travaille-pour » entre la classe « Personne » et la classe « Société » est une association « un-a-plusieurs » dans le cas où plusieurs personnes travaillent pour une société donnée et une seule société emploie une personne donnée. L'association « a-pour-capitale » est une association binaire « un-a-un ».

"1" signifie un exactement
"1..*" signifie de un à plusieurs
"*" et "0..*" signifient de zéro à plusieurs
"3..5" signifie l'intervalle 3 à 5 inclus
"2, 4, 18" signifie explicitement les valeurs 2, 4, 18

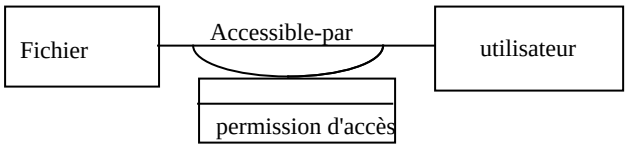
* = plusieurs
par défaut, une ligne simple = un

La multiplicité est écrite du côté de la classe destination du rôle.
Ne pas se préoccuper de la multiplicité trop tôt dans le développement.
La distinction majeure doit être faite entre un et plusieurs...

Attributs de lien

Un *attribut de lien* est une propriété de lien.

notation : une boîte rattaché par une boucle à l'association



Modéliser une association en classe

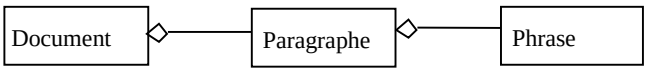
L'étape suivante est de modéliser une association sous forme de classe.

L'agrégation

L'*agrégation* est une relation « composé-composant » ou « partie de » dans laquelle les objets représentant les composants d'une chose sont associés à un objet représentant l'assemblage (ou l'agrégation) entier.

L'agrégation est transitive non symétrique.

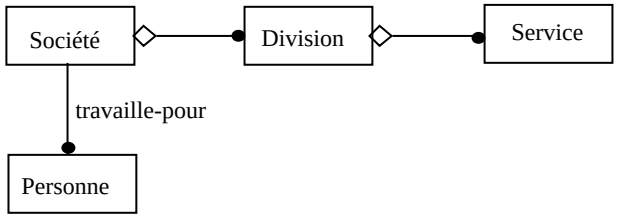
L'agrégation est une forme spéciale d'association. Le symbole représentant l'agrégation est le losange. Il est situé à côté de la classe composée.



La figure ci-dessus exprime l'idée qu'un document est composé de paragraphes, eux-mêmes composés de phrase.

Agrégation & association

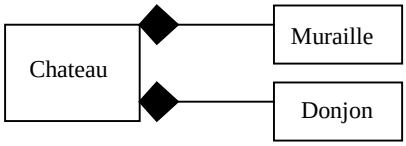
L'agrégation est un cas particulier d'association.



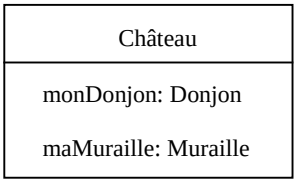
Agrégation et association

Composition & Agrégation

UML propose la notion de composition. Cette notion est très proche de celle d'agrégation. La composition est une agrégation *réalisée par valeur*. Elle se note avec un rectangle noir.



Tout se passe comme si les classes composantes étaient des attributs de la classe composée.



En fait cette distinction entre composition et agrégation est intéressante lors de la programmation mais rarement lors d'une spécification ou d'une conception.

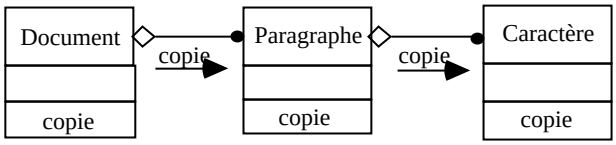
Agrégations récursives

fixe
variable : nombre fixe de niveaux mais nombre de parties par niveau variable
récursif : l'agrégat contient directement ou indirectement une instance de la même sorte que l'agrégat.

Propagation des opérations

Si on effectue une opération sur l'agrégat, cela *propage* des opérations de copie au niveau des objets parties de l'agrégat.

exemple:
Un document est composé de paragraphes.
Un paragraphe est composé de caractères.
document.copie se propage en paragraphe.copie qui se propage en caractère.copie



propagation des opérations

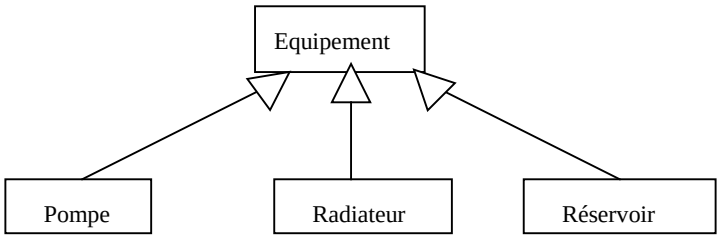
2.3. Généralisation, héritage, redéfinition

super-classe et sous-classes

La *généralisation* est la relation entre une classe et une ou plusieurs versions affinées de la classe.

On appelle la classe dont on tire les précisions, la *super-classe* et les autres classes les *sous-classes*.

La notation utilisée pour la généralisation est le triangle :



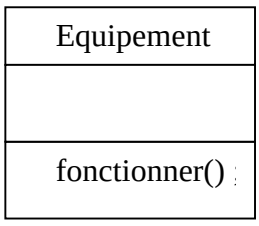
Dans cet exemple, la classe « Equipement » généralise les classes « Pompe », « Radiateur » et « Reservoir ».

La relation inverse de la généralisation est la spécialisation. Dans l'exemple, les classes « Pompe », « Radiateur » et « Reservoir » spécialisent la classe « Equipement ». La notation utilisée pour représenter la spécialisation en UML est aussi le triangle. La différence entre généralisation et spécialisation est faible ; ce n'est qu'une question de sens de lecture du diagramme : la généralisation suit la direction indiquée par le triangle et la spécialisation, la direction opposée.

Héritage et définition de propriétés

Si une propriété est définie dans la super-classe, la propriété existe alors dans toutes les sous-classes. On dit que la sous-classe *hérite* des propriétés de la super-classe.

Par exemple, si on définit la méthode fonctionner dans la classe Equipement :



Alors toute instance d'une sous-classe, par exemple une pompe, héritera de la méthode fonctionner et pourra donc appeler cette méthode.

Généralisation et héritage sont transitifs à travers un nombre arbitraire de niveaux.

Les notions de *ancêtre* et de *descendant* existent en UML ; elles correspondent aux notions habituelles.

Quand une instance utilise une propriété qui n'est pas définie dans la classe de l'instance, le langage objet remonte l'arbre d'héritage depuis la classe de l'instance jusqu'à la première classe où est définie la propriété.

La *classe d'origine* d'une propriété est la classe la plus générale où est définie la propriété.

Discriminant

Un *discriminant* correspond à un type de généralisation.

Exemple : la classe « Véhicule » peut être généralisée par le mode de déplacement (propulsion, moteur à essence, énergie musculaire, etc) ou par l'environnement de déplacement (air, eau, terre, etc). Le discriminant qui va de « Véhicule » à « Bateau » est « eau ».

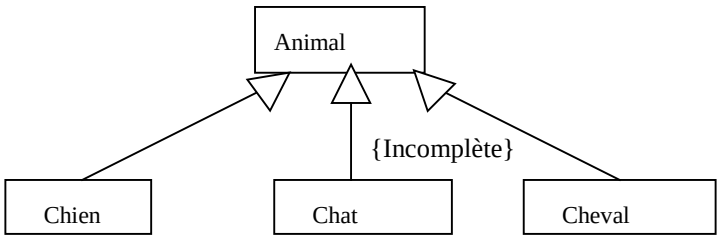
L'usage veut qu'une seule propriété doit être généralisée à la fois.

Utilisation de la généralisation

La généralisation facilite la modélisation en regroupant ce qui est commun aux classes de ce qui ne l'est pas. L'héritage des opérations aide lors de l'implémentation à réutiliser des classes.

Généralisation complète ou incomplète

Une généralisation peut être précisée *complète* ou *incomplète* grâce à un commentaire associé au diagramme. Par exemple, avec le diagramme de généralisation suivant :

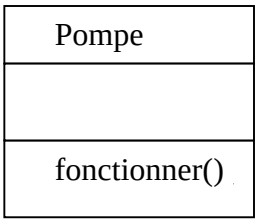


le commentaire **{Incomplète}** placé sous le diagramme exprime l'idée que des instances de la classe « Animal » sont des chats, des chiens, des chevaux *ou d'autres animaux*. Par contre, le commentaire **{Complète}** exprime l'idée qu'un animal *ne peut être autre chose* qu'un chat, un chien ou un cheval.

Redéfinition des propriétés

Une sous-classe peut *redéfinir* une caractéristique de la super-classe en définissant une propriété portant le même nom. La nouvelle propriété remplace celle de la super-classe pour les objets de la sous-classe. Une redéfinition ne pas changer la signature de la propriété.

Par exemple, si l'on définit la méthode fonctionner dans la classe « Pompe » :



Alors cette définition est en fait une RE-définition car la méthode fonctionner existe déjà dans la classe « Pompe » car elle est héritée de la classe « Equipement ». La redéfinition ne présente un intérêt que si l'on veut spécifier qu'une pompe fonctionne de manière spéciale par rapport aux autres équipements.

Agrégation & généralisation

L'agrégation est différente de la généralisation. Elle met en relation des *instances*.
La généralisation met en relation des *classes*.

agrégation = relation "partie de"
généralisation = relation "sorte de" ou "est un"

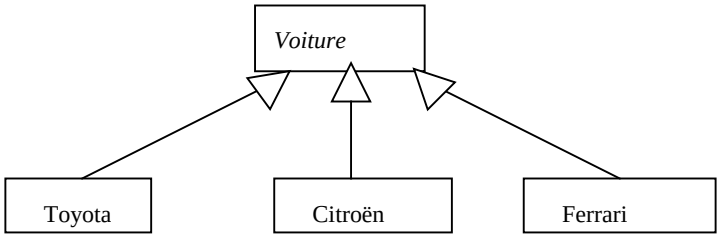
agrégation = relation "ET"
généralisation = relation "OU"

Une lampe est faite d'un socle ET d'une ampoule ET d'un abat-jour ET d'un interrupteur ET d'un câble
Une lampe est un lampe à incandescence OU une lampe à pétrole OU une lampe à fluorescence

2.4. Classes abstraites, instanciation

Une classe *abstraite* est une classe qui n'a pas d'instances directes. En UML, une classe abstraite est notée en *italique*. Une classe *concrète* est une classe qui a des instances directes. On appelle aussi une classe concrète, une classe *instanciable.*

Par exemple, la classe « Voiture ». Une « Toyota » est une « Voiture ». Une « Citroën » est une « Voiture ». Ma Toyota rouge est une instance directe de la classe « Toyota ». C'est une instance indirecte de la classe « Voiture ». La classe « Toyota » est « instanciable ». La classe « Voiture » ne l'est pas. La classe « Voiture » est une classe abstraite. La classe « Toyota » est une classe concrète.



L*'instanciation* est une relation qui relie une classe à une instance.

Dans un langage de programmation, l'instanciation correspond au moment ou le constructeur de la classe est appelé :

```
Personne p = new Personne(« Alice ») ; // en java
Personne * p = new Personne(« Alice ») ; // en C++
```

Dans cet exemple, la classe « Personne » est instanciable.

Instanciation & Spécialisation

Attention à ne pas confondre instanciation et spécialisation. On peut écrire :

a) Toyota est une Voiture.
b) Citroën est une Voiture.
c) Ma 2CV est une Citroën.
d) Ma Toyota est une Voiture.

a) et b) sont des *spécialisations*
b) et d) sont des *instanciations*

Pour différencier une instanciation d'une spécialisation il faut se demander si l'on est capable de remplacer le « est » (qui est un mot trop vague) par « est une sorte de » ou plutôt par « est un exemple de » (qui sont des mots plus précis). Si on est capable de remplacer le « est » plutôt par un « est une sorte de » alors on a une spécialisation. Et si on est capable de remplacer le « est » par un « est un exemple de » alors on a une instanciation.

2.5. Généralisation par extension ou par restriction

extension : une sous-classe ajoute des propriétés à la super-classe.
restriction : les valeurs prises par les propriétés sont restreintes.

L'appartenance à une classe peut être définie de deux façons:
implicitement par une règle
explicitement par énumération

Les opérations redéfinies

Redéfinir pour *restreindre* :
Exemple : le type d'un argument d'une opération d'une sous-classe peut se spécialiser. exemple : La classe Ensemble possède une opération ajouter(Element). La classe Entier hérite de la classe Element. La classe EnsembleEntiers hérite de la classe Ensemble. La classe EnsembleEntiers redéfinies l'opération ajouter en spécialisant le type Element en Entier.

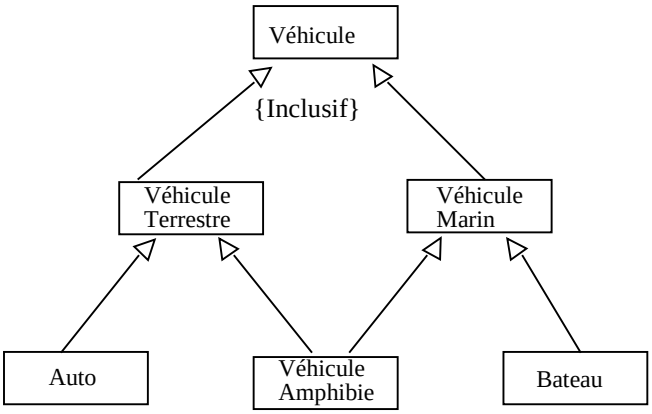
Redéfinir pour *optimiser*:
Exemple : l'opération ajouter(Entier) de la classe EnsembleEntier peut être optimisée si par exemple les ensembles d'entiers sont ordonnés.

Redéfinir pour *raisons pratiques*:

2.6. Héritage multiple

L'*héritage multiple* permet à une classe d'avoir plus d'une super-classe et d'hériter des propriétés de tous ses parents.

Une classe avec plusieurs super-classe est appelée une *classe de jointure*.



Héritage multiple

La classe « Vehicule Amphibie » hérite à la fois de la classe « Véhicule Terrestre » et à la fois de la classe « Véhicule Marin ». C'est la classe de jointure.

Afin d'exprimer que les sous-classes « Véhicule Terrestre » et Véhicule Marin » ne sont pas disjointes, UML prévoit d'insérer le commentaire **{Chevauchement}** ou **{Inclusif}** au niveau de la généralisation concernant ces deux classes.

En l'absence de commentaire sur une généralisation, cette généralisation est disjointe. Mais UML prévoit la possibilité de mettre explicitement le commentaire **{Exclusif}** ou **{Disjoint}** a côté d'une généralisation.

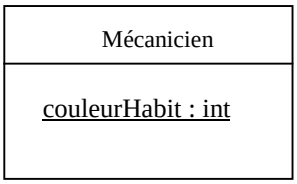
Généralisation & Héritage

Les concepts objet de généralisation et d'héritage sont étroitement liés mais elle sont différente. La généralisation est une relation conceptuelle entre plusieurs classes alors que l'héritage est un mécanisme qui découle du concept de généralisation. L'héritage est un mécanisme qui est implémenté dans les langage orienté objet dont le programmeur n'a pas à se soucier.

2.7. Propriétés de classe ou propriété d'instance ?

UML, comme les autres méthodes orientées objet, propose une distinction entre propriétés *de classe* et propriété *d'instance*. Une propriété *de classe* est un propriété dont la valeur est identique pour toutes les instances de la classe et une propriété *d'instance* est une propriété dont la valeur est différente pour chaque instance.

Par exemple, si dans un garage tous les mécaniciens sont habillés en bleu, alors l'attribut « couleurDeTravail » de la classe « Mecanicien » sera définie en attribut *de classe*. En UML, une propriété *de classe* est soulignée. (En Java ou en C++, on utilise le mot-clé 'static', en OMT on utilise '$').



Par contre si les mécaniciens sont habillés de couleurs différentes, alors l'attribut « couleurDeTravail » de la classe « Mecanicien » sera impérativement définie en attribut *d'instance*.

Puisque leur valeur est commune à toute la classe, les propriétés de classe sont valorisées une seule fois dans la classe et sont communes à toutes les instances.

Toutes les méthodes classiques de gestion des instances d'une classe telles que :

void detruireToutesLesInstances()
Instance deIHMACreationDUneInstance()
void deIHMADestructionDUneInstance()
Instance getInstance(String)

seront judicieusement placées en méthodes *de classe*.

Dans une classe, il peut être pratique de placer certaines informations de la classe en attributs *de classe*. Par exemple, la liste des instances d'une classe. Si on choisit de la placer dans la classe elle-même, elle sera un attribut *de classe*. Mais on peut aussi préférer placer la liste des instances dans la classe qui utilise la classe ou dans une instance composite qui utilise la classe ; dans ce cas, la liste des instances est un attribut *d'instance* de la classe utilisatrice.

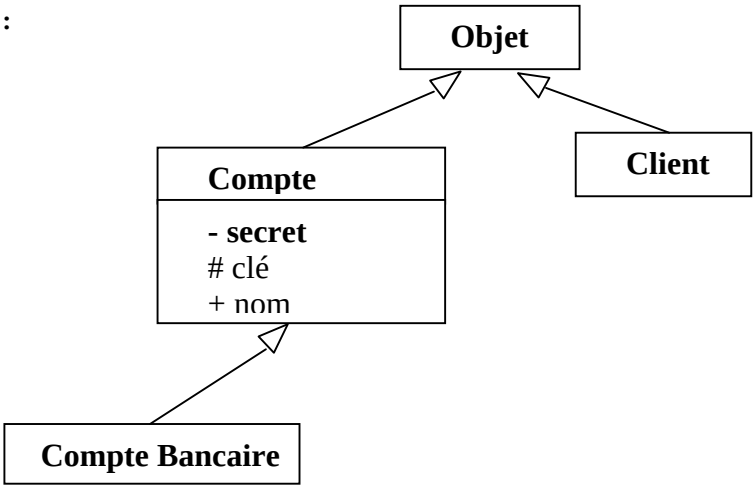
2.8. Visibilité des propriétés, principe d'encapsulation des données

UML, comme les autres méthodes orientées objet, propose une distinction entre propriétés *privées, protégées* et *publiques*. Une propriété *privée* est un propriété visible de l'intérieur de la classe et invisible de l'extérieur. Une propriété *protégée* est un propriété visible de l'intérieur des sous-classes de la classe et invisible d'ailleurs. Une propriété *publique* est un propriété visible de partout.

Notation UML

Une propriété publique est précédée du signe +. Une propriété privée du signe – et une propriété protégée du signe #.

Exemple :



Dans cet exemple, l'attribut « secret » de la classe « Compte » est privé : il n'est visible que depuis la classe « Compte ». L'attribut « clé » est protégé : il est visible de la classe « Compte » et de la classe « CompteBancaire » qui est sa sous-classe. Enfin, l'attribut « nom » est publique donc visible de partout : des classes « Compte », « CompteBancaire » et « Client ». (Il est également visible depuis la classe « Objet » mais il est maladroit d'utiliser une sous-classe dans une classe.)

Principe d'encapsulation

Le principe d'encapsulation des données est un mécanisme *qui oblige à rendre privés les attributs et publiques les méthodes*. Ce principe, un peu brutal, offre l'avantage de pouvoir faire évoluer le logiciel plus facilement lorsque la spécification de l'accès aux attributs changera (il suffira alors de ne changer que le corps des méthodes d'accès aux attributs), ou bien en programmation concurrente lorsque plusieurs tâches sont susceptibles d'accéder aux données simultanément. Mais, contrairement à ce que la plupart des livres de programmation objet contiennent, nous déconseillons le suivi de ce principe aux débutants en programmation objet. La métapropriété d'une propriété d'être de classe ou d'instance (cf paragraphe précédent) est, selon nous, beaucoup plus importante que la métapropriété de visibilité. Heureusement, la syntaxe UML offre la souplesse de spécifier la visibilité, privée, protégée ou publique pour chaque propriété et ainsi de ne pas obliger le modélisateur à suivre le principe d'encapsulation.

2.9. Règles et conseils

. Ne pas mettre de flèches pour représenter les associations dans les diagrammes de classes. Les symboles > et < sont prévus à cet effet pour indiquer le sens correct de l'association.

. Ne pas mettre des S aux noms de classe.

. Pas de pointeurs ou références vers d'autres objets sous forme d'attribut : utiliser les associations.

. Un diagramme de classes ne doit pas contenir deux fois la même classe.

. Deux classes peuvent être associées par plusieurs associations différentes.

. Respecter absolument la correspondance entre le diagramme d'instances et le diagramme classes.

. Séparer les diagrammes d'instances, de classes et de généralisation.

. Il est possible de mettre des agrégations sur un diagramme classes.

. Il est possible et conseillé de faire plusieurs diagrammes classes, un par point de vue que l'on veut exprimer.

. Garder un modèle simple aussi simple que possible : pas de complications.

. Choisir les noms avec soin. Attention aux sens multiples d'un mot. Le choix des bons noms est la facette déterminante de la modélisation objet.

. Eviter si possible les associations ternaires ou n-aires ; surtout lorsque l'on se rapproche de la phase de codage.

. Eviter les généralisations profondément imbriquées.

. Le modèle objet ne se trouve pas du premier coup. Il doit être révisé plusieurs fois avant d'être dans une version correcte et stable. Soumettre le modèle à des avis extérieurs.

. Documenter les modèles objets avec les raisons commentées qui guident les choix effectués.

. Ne pas exprimer les multiplicités trop tôt.

. A partir d'un ensemble d'exemples sur un domaine il est possible de trouver des multiplicités différentes. Si on généralise strictement les exemples, les multiplicités sont précises. Si l'on utilise nos connaissances sur le monde réel, les multiplicités sont moins précises.

. Un diagramme de classes ne doit pas, dans la mesure du possible, contenir une classe et une de ses spécialisations.

2.10. Sur-spécification des classes

Les erreurs courantes en orienté objet peuvent être causées par une sur-spécification des classes; ce qui signifie avoir trop de classes. Ce paragraphe répertorie les principales erreurs dues à une surspécification.

Nommer une classe GestionT

Si dans le modèle objet, apparaît une classe GestionT et une classe T, c'est une erreur. En effet, une classe est déjà une entité informatique qui faite pour gérer (des instances). Donc écrire « classe getsionT » est un euphémisme. Pour supprimer cette erreur, supprimer la classe GestionT.

Nommer une classe BParticulier

Une classe est une entité « globale ». Une instance désigne une entité « particulière ». Donc appeler une classe BParticulier est maladroit. Supprimer la classe Bparticulier et remplacer par la classe B si elle n'existe pas déjà.

Eviter si possible les noms de classe composés NomAdjectif

Eviter si possible les noms de classes composés d'un nom et d'un adjectif : exemple « CompteBancaire », « TrainDemandé », surtout quand la classe « TrainDemandé » n'hérite pas de « Train », car dans ce cas, un « TrainDemandé » n'est pas un « Train » ; ce qui est troublant pour le sens commun…

Par contre, si la classe « NomAdjectif » hérite de la classe « Nom » alors il n'y a pas d'ambiguïté et le problème n'existe pas.

Interdire les changements de classe au cours de la vie d'une instance

Si les classes « CompteDebiteur » et « CompteCrediteur » héritent de la classe « Compte », cela crée un gros problème. En effet, une instance d'une classe garde le même type, est toujours une instance de la même classe, au cours de toute sa vie d'instance ; c'est à dire entre sa création et sa destruction. Une instance ne peut être créée dans la classe « CompteDebiteur », puis transférée vers la classe « CompteCrediteur » lorsque un client va remplir son compte ! C'est possible dans la vie réelle mais pas en technologie orientée objet ! Pour gérer ce cas, le montant du compte donnera implicitement l'information sur le fait que le compte est débiteur ou créditeur.

Ne pas remettre dans la sous-classe, les attributs ou méthodes hérités de la super-classe

Quand on ne veut pas les redéfinir, il ne faut pas remettre les mêmes attributs ou méthodes dans la classe héritée (la sous-classe) et dans la classe générale (la super-classe): en objet, le principe de l'héritage est de définir les choses une seule fois à un seul endroit !

Si on veut signifier qu'il est redéfini, on remet à nouveau l'attribut ou la méthode dans la classe héritée.

2.11. Placement des informations

Placer les informations au bon endroit

Il faut placer les attributs et les méthodes dans la classe adéquate : ne pas placer les infos sur un client dans la classe mécanicien.

Ne pas mettre un attribut sur le type de l'objet

Une classe ne doit pas contenir un attribut sur le type de l'instance créée. Si c'est le cas, essayer de spécialiser cette classe en plusieurs sous-classes.

Mettre les informations de liste d'instances en attribut de classe ou pas ?

Placement en attribut de classe d'une classe ou bien en attribut d'instance d'une classe utilisatrice de la classe : il faut choisir une façon de placer les listes d'objet. Chacune des deux méthodes a son avantage. En choisir une.

Mettre les noms de rôle des associations ou pas ?

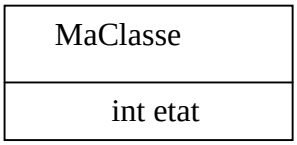
En phase de spécification des besoins ou en conception générale, en OMT, il ne faut pas mettre les associations sous forme d'attribut dans les rectangles décrivant les classes. En conception détaillée ou en codage, on les met avec les noms de rôle des associations (monClient, monMécanicien seront deux rôles des classes Mécanicien et Client si celles ci sont associées)

3. Le modèle des états et le modèle d'interaction

La partie précédente décrivait le formalisme modélisant statiquement un domaine, à savoir le modèle des classes. Ce modèle ne permettait pas de modéliser l'aspect dynamique d'un domaine. Le *modèle des états* et le *modèle d'interaction* permettent la modélisation du point de vue dynamique. Il modélise l'évolution des objets au cours du temps. Le modèle des états comprend les *diagrammes d'états-transitions*. Le modèle d'interaction comprend les *diagrammes de collaboration* et les *diagrammes de séquence*.

Etat

Un objet possède un *état* à un instant donné. L'état de l'objet est une notion durable à l'échelle de temps d'évolution des objets. Par exemple, si l'objet est un dé à jouer, on peut dire que le dé possède 6 états possibles et que le dé posé sur la table est dans un état durable (tant qu'on ne le lance pas). L'état peut être spécifié explicitement:



L'état peut aussi ne pas être explicitement présent sous forme d'un attribut. Dans ce cas, l'état de l'objet *au sens strict* correspond à l'ensemble des attributs de l'objet. Et l'état *au sens large* correspond à l'ensemble des attributs et liens de l'objet.

Evénements

Un *événement* est un stimuli externe ou interne à l'ensemble des objets. Il se produit instantanément à l'échelle d'évolution du système. Un événement est noté entre guillemets. Par exemple : « Le vol AF-123 part de Chicago ».

Un événement peut être *reçu* ou *envoyé* par un objet. Quand l'objet est spécifié par le contexte, on note avec une flèche vers le bas un événement reçu par l'objet et une flèche vers le haut un événement envoyé par l'objet. Par exemple, voici un « événementReçu » ↓, et voici un « événementEnvoyé »↑. Si l'objet n'est pas spécifié, les flèches n'ont pas de signification car un même événement peut être envoyé par un objet et reçu par un autre.

Les événements peuvent être hiérarchisés dans un diagramme de généralisation. Par exemple :

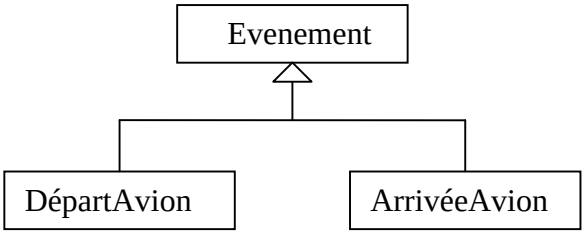


Diagramme de séquence

Pour commencer à décrire l'évolution d'un ensemble d'objets, il est possible de dessiner d'abord un *diagramme de séquence*.

Horizontalement, on place les instances concernées par un scénario et on relie les instances par des flèches indiquant le flux d'événements. Verticalement, le temps est représenté.

Ce type de diagramme donne une première idée des événements qui pourront être pertinents dans la modélisation. On répète ce diagramme autant de fois qu'il existe de scénarii d'événements possibles

Exemple du téléphone: Dans le suivi d'événements ci-dessous, un utilisateur appelant décroche le téléphone qui envoie un signal sur la ligne téléphonique et la tonalité à l'utilisateur appelant :

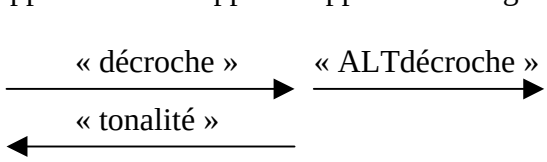
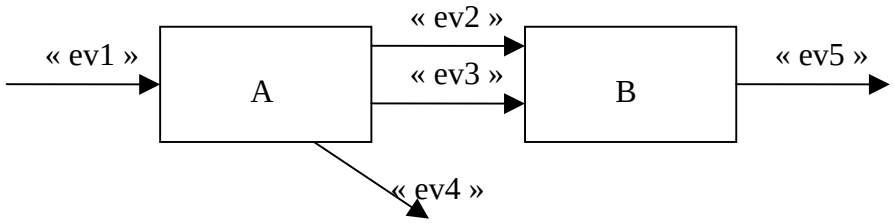


Diagramme de collaboration

Un diagramme de flux ou *diagramme de collaboration* est un diagramme sur lequel les classes d'objets sont représentées avec des rectangles reliés par des flèches représentant les flux d'événements entre les classes.

exemple :



Ici, l'événement « ev1 » est recevable par la classe 'A'. Les événements « ev2 », « ev3 » et « ev4 » sont émissibles par 'A'. Les événements « ev2 » et « ev3 » sont recevable par la classe 'B'. Enfin, l'événement « ev5 » est émissible par la classe 'B'.

Ce type de diagramme sert à répertorier tous les événements, reçus et envoyés, relatifs à chaque classe.

Diagramme d'état-transitions :

Un diagramme d'état est un graphe dont les nœuds sont les valeurs possibles de l'état de l'objet et les arcs sont les transitions entre ces valeurs.

Par abus de langage dans la suite du document, on dit que les nœuds du graphe sont les états de l'objet au lieu de dire les valeurs des états de l'objet.

Le diagramme d'états est le diagramme que l'on cherche à obtenir lorsque l'on modélise la dynamique des objets. Il peut être obtenu soit directement dans les cas simples, soit indirectement dans les cas complexes, après avoir faits les diagrammes de suivi d'événements et/ou les diagrammes de flux d'événements.

Sur un diagramme d'états, on place les événements reçus et envoyés par l'objet et les transitions associées. On place aussi les *actions* et les *activités*. Il existe plusieurs types d'actions : *entrée*, *sortie*, *interne* ou de *transition*.

Les actions et activités

Une action est supposée avoir une durée nulle à l'échelle d'évolution des objets.
Une activité par contre a une durée non nulle à l'échelle d'évolution des objets.

Une action est effectuée à la suite de la réception d'un événement (notion instantanée) alors qu'une activité est effectuée pendant que l'objet est dans un état (notion durable).

Une activité est spécifiée avec le mot-clé `do :`.

Action d'entrée

Une action d'entrée est effectuée lorsque l'on rentre dans un état.
Elle est annoncée avec le mot-clé `entry :`.

Action de sortie

Une action de sortie est effectuée lorsque l'on sort d'un état.
Elle est annoncée avec le mot-clé `exit :`.

Action interne

Une action interne est effectuée lorsque l'objet reçoit un événement sans faire changer l'état de l'objet. Elle est annoncée avec le mot-clé `on` suivi du nom de l'événement entre guillemets.
exemple : `on « evenement » :`.

Action de transition

Une telle action est une action associée à une transition. Dans ce cas, une *condition* ou *garde* peut être associé(e) à la réception de l'événement associé à la transition. le garde est spécifié entre crochets [] et l'action est précédée du /.

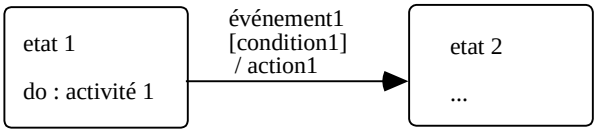
Par exemple, dans la phrase suivante : « Quand je sors le matin, si la température est glaciale, je mets mes gants. », on a les correspondances suivantes

"Quand je sors le matin,	(événement)
si la température est glaciale,	(condition)

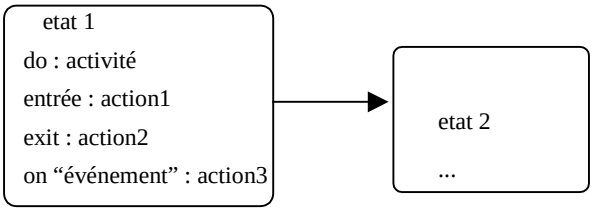
je mets mes gants." (action de transition)

Elle est formulée avec « `événement` » ↓ `: [condition] / action`

Par exemple, le diagramme suivant exprime une transition entre deux états avec une activité ‘activité1’ associée à l’état ‘état1’, une action de transition ‘action1’, une condition ‘condition1’, sur réception de l’événement ‘événement1’.



Sur l’exemple suivant, l’activité ‘activité’ est effectuée pendant que l’objet est dans l’état ‘état1’ ; en entrée de l’état ‘état1’, l’action ‘action1’ est effectuée ; en sortie, l’action ‘action2’ est effectuée ; enfin, si l’événement ‘événement’ est recçu dans l’état ‘état1’ alors l’action interne ‘action3’ est effectuée.



Une action interne est différente d'une transition réflexive. En effet, cette dernière entraîne l'exécution de l'action de sortie de l'état juste avant et l’exécution de l’action d’entrée juste après.

Lorsque plusieurs transitions arrivent sur un état, le choix du placement d’une action sur une transition ou bien en entrée de l’état doit être fait avec précision. De même, lorsque plusieurs transitions partent d’un état, le choix du placement d’une action sur une transition ou bien en sortie de l’état est déterminant.

Transition automatique

Une transition *automatique* est une transition qui se déclenche à la fin d’une activité de l’objet. Elle est dite automatique car elle se déclenche sans que l’objet ne reçoive d’événement. Une transition automatique peut avoir un garde et une action associés.

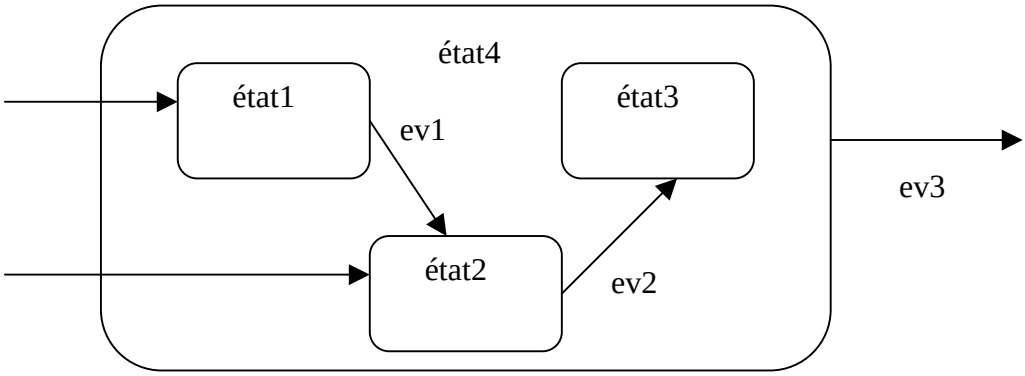
Etat initial et états finaux

L’état *initial* d’un objet est spécifié avec un rond noir plein et les états *finaux* avec des ronds noirs encerclés.

Hiérarchie d'états

En cours de modéliation avec un diagramme d'états, il peut être judicieux de regrouper plusieurs états en un seul ou bien de décomposer un état en plusieurs sous-états.

Exemple :



Ici, l'état 'état4' est un sur-état des états 'état1', 'état2', 'état3'. Les deux transitions qui rentrent dans l'état 'état4' spécifient dans quel sous-état on arrive. La transition sortante de l'état 'état4' exprime que, quel que soit le sous-état dans lequel l'objet se trouve, si l'événement 'ev3' arrive, alors l'objet sort de l'état 'état4'.

On peut bien sûr placer des actions d'entrée ou de sortie sur des sur-états ou des sous-états.

Relations entre le modèle objet le modèle dynamique

Le modèle dynamique spécifie les séquences acceptables de modification d'objet.

Le modèle dynamique d'une classe est hérité dans les sous-classes ; il est possible de redéfinir un diagramme d'états dans une sous-classe. Ce qui est naturel puisque un état d'objet est un attribut d'objet (donc héritable et redéfinissable).

Toutes les classes ne nécessitent pas de diagramme d'états

Selon les domaines à modéliser, on peut commencer ou non par faire les scénarios d'événements et les diagrammes de flux avant de faire les diagrammes d'états.

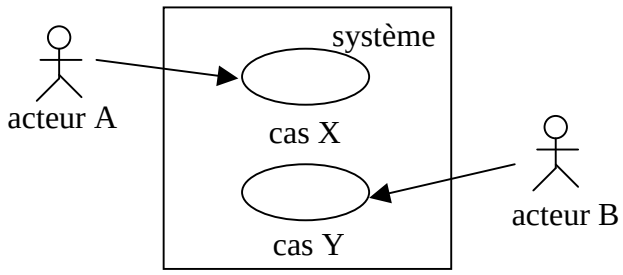
4. Le modèle des cas d'utilisation

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. Avant UML, ils n'étaient pas formalisés par les autres méthodes objet telles que OMT.

Les cas d'utilisation sont utiles lors de l'élaboration du cahier des charges ou du document de spécifications des besoins du logiciel.

Le *modèle des cas d'utilisation* comprend les *acteurs*, le *système* et les *cas d'utilisation*.
L'ensemble des fonctionnalités du système est déterminé en examinant les besoins de chaque acteur, exprimés sous forme de famille d'interactions dans les cas d'utilisation.

Les acteurs se représentent sous forme de petits personnages qui déclenchent les cas. Ces derniers se représentent par des ellipses contenues dans un rectangle représentant le système.



Dans cet exemple, l'acteur A déclenche le cas X et l'acteur B déclenche le cas Y.

Il existe quatre catégories d'acteurs :
- les acteurs principaux,
- les acteurs secondaires,
- le matériel externe,
- les autres systèmes.

Chaque acteur doit être décrit en 3 ou 4 lignes de manière claire et concise.

Un cas d'utilisation décrit un ensemble de scénarios du point de vue de l'utilisateur grâce a des diagrammes de séquence ou des diagrammes de collaboration.

Les relations entre cas d'utilisation

relation de communication — déclenche

relation d'utilisation — <<Utilise>>

relation d'extension — <<Etend>>

Règles de mise en œuvre

La description du cas d'utilisation comprend les points suivants :

le *début* du cas exprimé par : « les cas débute quand X se produit »,
la *fin* du cas exprimé par : « les cas se termine quand Y se produit »,
l'*interaction* entre le système et les acteurs qui décrit clairement la frontière du système,
les échanges d'informations
la chronologie et l'origine des informations utilisant :
les *diagrammes de séquence*
ou les *diagrammes d'activités.*
les répétitions de comportement du type :

```
boucle
      quelquechose
fin de boucle
```

ou :

```
pendant que…
      autre chose
fin pendant
```

les situations optionnelles

Diagrammes d'activités

Les diagrammes d'activités suivent les même formalisme que les diagrammes d'état-transitions sauf que les états sont remplacés par des activités, avec la possibilité pour les activités de se synchroniser.

5. De UML vers le codage (C++ ou Java)

Ce paragraphe traite du passage d'une conception générale ou détaillée faite avec UML vers le codage en C++ ou en Java.

Classe, instance, attribut, méthode

Une classe, une instance, un attribut ou une méthode UML sera une classe, une instance, un attribut ou une méthode en Java et une classe, une instance, un membre ou une fonction membre en C++. Il n'y a pas d'ambiguité. Le passage est un-à-un.

Attributs et méthodes « de classe » ou « d'instance »

UML, C++ et Java font la distinction entre les attributs ou méthodes « de classe » ou « d'instance ».

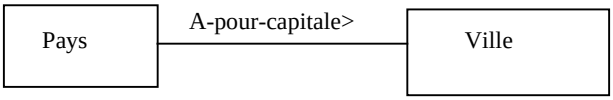
La correspondance est la suivante :

	UML	Java	C++
attribut, méthode « de classe »	souligné	static	static
attribut, méthode « d'instance »	-	-	-

Association

Pour les associations UML, il y a plusieurs façons de faire. Mais dans tous les cas, une association UML se traduira par un ou plusieurs attributs. Ces attributs seront des pointeurs ou des références vers des objets ou des listes.

Exemple d'une association un-à-un :



En C++ on écrira :

```
class Pays {…
    Ville * aPourCapitale ; …
} ;

class Ville {…
    Pays * estLaCapitaleDe ;…
} ;
```

Ainsi, les deux rôles de l'association UML seront traduits par deux pointeurs C++.

En Java, on écrira :

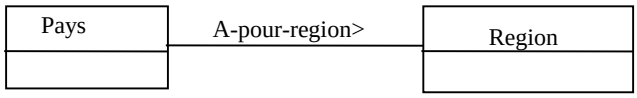
```
class Pays {…
    Ville aPourCapitale ;…
}

class Ville {…
    Pays estLaCapitaleDe ;…
}
```

Ainsi, les deux rôles de l'association UML seront traduits en deux références Java.

On remarquera que cette façon de faire est complète mais que toute relation du modèle UML ne se traduit pas nécessairement en un couple de pointeurs ou références au niveau du codage. On peut très bien n'implémenter qu'un seul sens de l'association UML en ne prenant qu'un seul pointeur ou une seul référence.

Cette façon de faire marche bien pour les associations dont la multiplicité est 0 ou 1. Pour les associations un-à-plusieurs, le programmeur a besoin d'autre chose.

Exemple d'une association un-à-plusieurs :



Dans ce cas, un pays ayant plusieurs région, on a besoin d'un attribut pouvant contenir un nombre quelconque d'éléments. Un attribut de type tableau n'est pas adapté car il possède une taille qui limite le nombre d'éléments. Un attribut étant de la classe Liste conviendra parfaitement . En C++, on écrira par exemple :

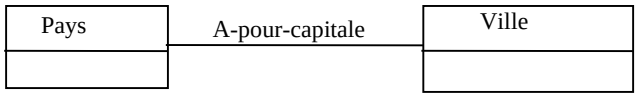
```
class Pays {…
    Liste * aPourRegions ;…
} ;

class Region {…
    Pays * estUneRegionDe ;…
} ;
```

En synthétisant les deux exemples précédents sur les pays, les villes et les régions, on peut dire que les attributs monovalués (un attribut qui contient au plus une valeur) de UML seront traduits en pointeurs ou références sur des objets au niveau du codage C++ ou Java et que les attributs multivalués (un attribut qui peut contenir un nombre quelconque de valeurs) seront traduits en listes d'objets, ou à la limite en tableaux d'objets.

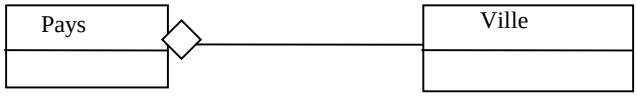
Traduire une agrégation UML : par référence ou par contenance ?

Les relations d'agrégation et de composition UML se traduisent au niveau du codage en C++ ou en Java.

A première vue, en C++, on peut faire la distinction grossière suivante :



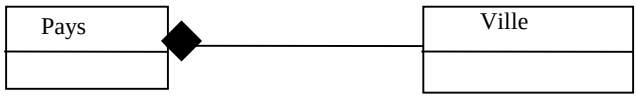
ou :



sera traduit par :

```
class Pays {…
    Ville * aPourCapitale ;…
} ;

class Ville {…
    Pays * estLaCapitaleDe ;…
} ;
```

et la composition:



sera traduite au niveau de la classe Pays par :

```
class Pays {…
    Ville aPourCapitale ;…
} ;
```

En C++, cela signifie que l'objet Ville est contenu dans l'objet Pays. Cela traduit le fait qu'un Pays est composé de Ville. Donc le C++ permet de traduire la distinction (agrégation – composition) d'UML en une distinction (pointeur - objet contenu).

En Java, tout est référencé il n'y a pas de pointeur et la distinction [objet contenu, objet référencé] n'existe pas. Il n'y a pas d'objet contenu en Java : que des objets référencés. On écrira toujours :

```
class Pays {…
    Ville aPourCapitale ;…
}
class Ville {…
    Pays estLaCapitaleDe ;…
}
```

que l'association soit une agrégation ou pas.