Formation UML– Les diagrammes UML

El Hadji Gaye

Auteur	El Hadji Gaye
Pour	Formations
Date	15/01/2024
Objet	Les différents diagrammes UML.

I) Vocabulaire

- UML : c'est l'acronyme anglais pour « Unified Modeling Language ». On le traduit par « Langage de modélisation unifié ».

- OMG : Object Management Group.

II) Introduction

UML est basé sur l'approche par objets. Celle-ci vit le jour bien avant UML dans le domaine des langages de programmation. Simula, le tout premier langage à objets, est né dans les années 1960. Ce langage connut beaucoup de successeurs comme Smalltalk, C++, Java ou encore récemment C#.

Avec un langage de programmation, la description des objets est réalisée de façon formelle selon une syntaxe rigoureuse. Cette syntaxe présente l'inconvénient de ne pas être lisible par des non-programmeurs et d'être assez lente à déchiffrer même pour des programmeurs. Les humains, à la différence des machines, préfèrent les langages graphiques pour représenter des abstractions car ils peuvent ainsi les maîtriser plus facilement et plus rapidement, et obtenir une vue d'ensemble d'un système en un temps beaucoup plus court.

III) Historique de l'UML

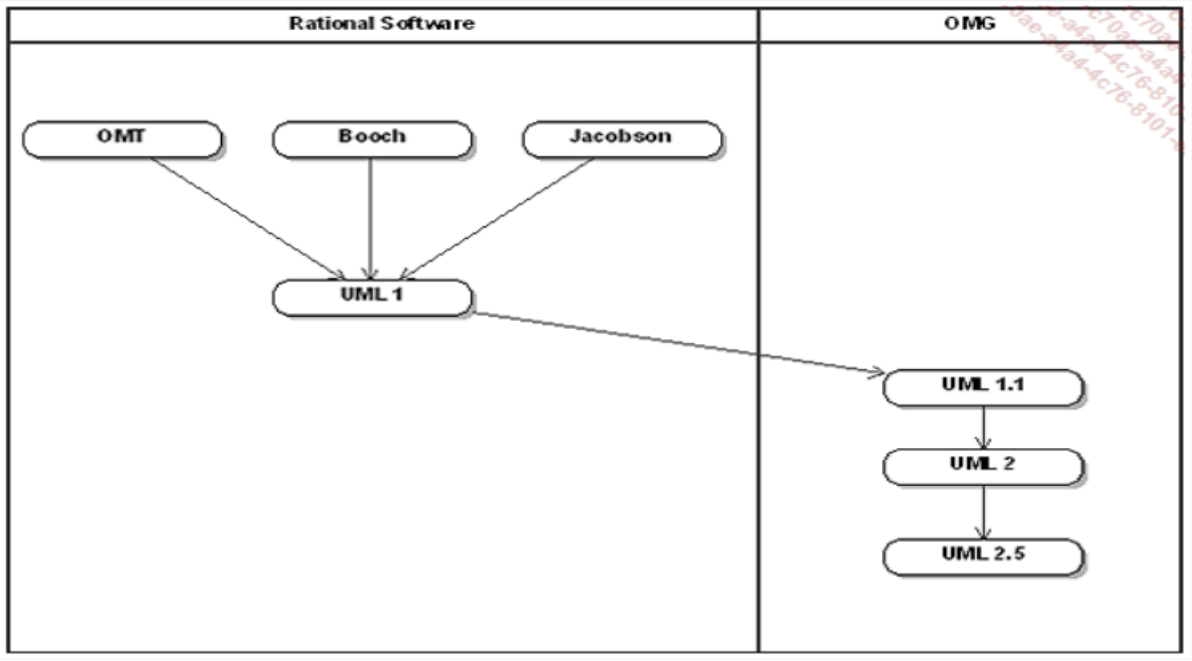
Dans les années 1980 et au début des années 1990, les notations graphiques se multiplièrent, chacune utilisant bien souvent sa propre notation. En 1994, Jim Rumbaugh et Grady Booch décidèrent de se regrouper pour unifier leurs notations. Celles-ci provenaient de leurs méthodes : OMT pour Jim Rumbaugh et la méthode Booch pour Grady Booch. En 1995, Yvar Jacobson décida de rejoindre l'équipe dite des Trois Amigos. Cette équipe travailla alors au sein de Rational Software.

La **version 1.0** d'UML fut publiée en 1997. Le travail d'évolution de la notation était alors devenu trop important pour trois personnes. Les Trois Amigos demandèrent ainsi l'aide de l'Object Management Group (OMG), un consortium de 800 sociétés et universités travaillant dans le domaine des technologies de l'objet. La notation UML fut adoptée par l'OMG en novembre 1997 dans sa version 1.1. L'OMG créa en son sein une Task Force chargée de l'évolution d'UML.

Cette Task Force a mis à jour UML plusieurs fois. En mars 2003, la version 1.5 ajouta la possibilité de décrire des actions grâce à une extension d'UML appelée Action Semantics, ou sémantique des actions.

La **version 2.0** fut publiée en juillet 2005. Elle constitue la première évolution majeure depuis la sortie d'UML en 1997. De nouveaux diagrammes ont été ajoutés et les diagrammes existants ont été enrichis de nouvelles constructions. Depuis juillet 2005, cette version 2.0 a été améliorée. En février 2009, la **version 2.2** a été publiée incluant la taxinomie officielle des profils UML. La **version 2.5** est publiée en juin 2015 et apporte principalement la possibilité de présenter les attributs et méthodes hérités d'une classe. La dernière version est la **version 2.5.1** publiée en 2017.

À l'aide d'un diagramme d'activités ci-dessous illustre l'évolution d'UML et en retrace la genèse ainsi que les principales versions.

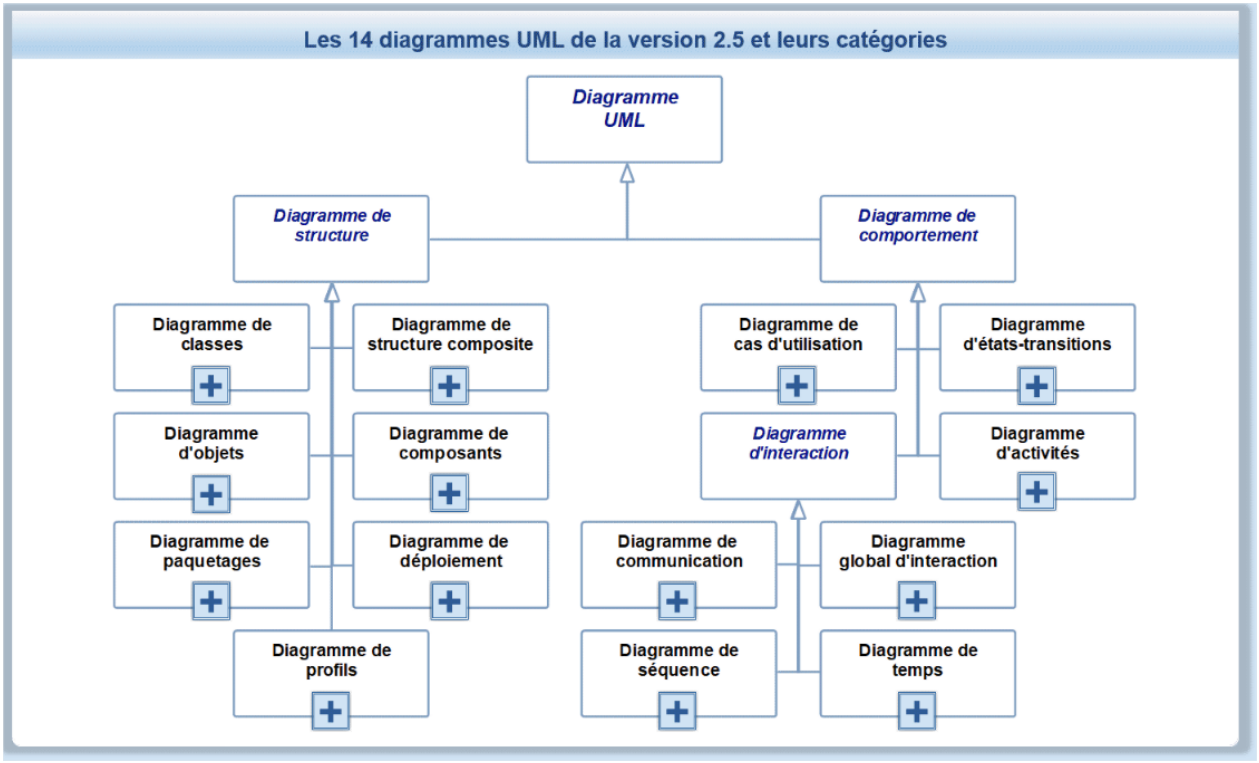


Dans la **version 2.5.1** du language UML publié par OMG (Object Management Group) en décembre 2017, definit et spécifie 14 diagramme standards.

- 7 diagrammes qui servent à décrire les aspects structurels d'un système informatique (logiciel et matériel).

- 7 diagrammes qui servent à décrire les aspects dynamiques (comportement) du système.

Voici ci-dessous un schéma illustrant les 14 diagrammes :



IV) Environnement technique

Dans le cas de cette formation vous aurez besoins d'installer le logiciel Dia dans votre machine avec l'URL : http://elhadji-gaye.fr/Installations/Dia-UML/dia_version_0.97.2_fr.zip

V) Les cas d'utilisation (Use Case)

Nous allons dans cette partie du cours vous faire découvrir les cas d'utilisation employés pour décrire les exigences fonctionnelles attendues, lors de la rédaction du cahier des charges d'un système, ou les fonctionnalités d'un système existant.

L'ensemble des cas d'utilisation d'un système contient les exigences fonctionnelles attendues ou existantes, les acteurs (les acteurs décrivent le rôle que prennent les utilisateurs du système) ainsi que les associations qui unissent acteurs et fonctionnalités. Cet ensemble détermine également les frontières du système, à savoir les fonctionnalités remplies par le système et celles qui lui sont externes.

Les cas d'utilisation servent de support pour les étapes de modélisation, de développement et de validation. Ils constituent un référentiel du dialogue entre les informaticiens et leurs clients et, par conséquent, une base pour l'élaboration au niveau fonctionnel du cahier des charges.

1. Description des cas d'utilisation

Les cas d'utilisation décrivent sous la forme d'une liste d'actions et d'interactions le comportement du système étudié du point de vue des acteurs. Ils définissent les limites du système et ses relations avec son environnement.

Cette définition doit être complétée. En effet, elle ne précise pas si un cas d'utilisation doit décrire l'intégralité ou une partie du dialogue entre un acteur et le système. Elle peut être formulée ainsi :

"Entre un acteur et le système, un cas d'utilisation décrit les actions et interactions liées à un objectif fonctionnel de l'acteur."

Un cas d'utilisation détaille la partie des exigences fonctionnelles du système concernant l'un des objectifs d'un acteur.

Exemple :

Considérons comme système un élevage de chevaux. L'achat d'un cheval par un client constitue un cas d'utilisation.

2. *Les acteurs*

Un même utilisateur externe du système peut prendre différents rôles vis-à-vis du système. Seule la notion de rôle est retenue en UML.

Un acteur décrit le rôle qu'un utilisateur externe du système prend lors d'une interaction avec le système.

Cette définition est étendue aux autres systèmes qui interagissent avec le système. Ils forment autant d'acteurs qu'ils prennent de rôles.

Deux catégories d'acteurs doivent être distinguées :

- Les acteurs primaires, pour lesquels l'objectif du cas d'utilisation est essentiel et constitue un objectif de l'acteur.
- Les acteurs secondaires, pour lesquels l'objectif du cas d'utilisation n'est pas essentiel bien qu'ils interagissent avec lui.

Exemple :

Reprenons l'exemple précédent du cas d'utilisation de l'achat d'un cheval par un client. L'acheteur d'un cheval est un acteur primaire. Les haras nationaux qui enregistrent le certificat de vente constituent un acteur secondaire.

3. Les scénarios

Un scénario est une instance d'un cas d'utilisation dans laquelle toutes les conditions relatives aux différents événements ont été fixées. Il n'y a donc pas d'alternatives lors du déroulement.

À un cas d'utilisation donné correspondent plusieurs scénarios.

Comme une classe qui détient les aspects communs de ses instances, un cas d'utilisation décrit de façon commune l'ensemble de ses scénarios en utilisant des branchements conditionnels pour représenter les différentes alternatives.

Exemple :

L'achat de Jorphée par Fien constitue un exemple de scénario du cas d'utilisation d'achat d'un cheval. Toutes les alternatives du déroulement sont connues, car Fien a acquis Jorphée.

4. L'association entre un acteur et un cas d'utilisation

L'association entre un acteur et un cas d'utilisation indique que cet acteur a la capacité d'interagir avec le système de la manière décrite par le cas d'utilisation.

Cette association est représentée graphiquement par un simple trait. Il est possible de l'orienter pour indiquer l'extrémité qui initie l'interaction avec l'autre partie. Cette orientation est réalisée à l'aide d'une flèche qui part de l'extrémité qui envoie les demandes vers celle qui les reçoit.

Exemple :

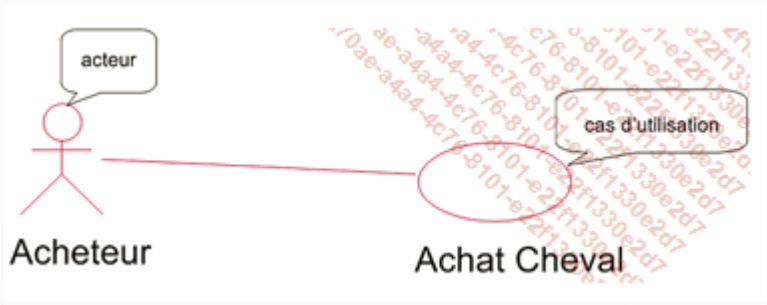
L'association qui lie l'acteur **Acheteur** au cas d'utilisation **Achat Cheval** indique que cet acteur détient la capacité d'acheter un cheval en interagissant avec ce cas d'utilisation.

5. *Le diagramme des cas d'utilisation*

Le diagramme des cas d'utilisation montre les cas d'utilisation représentés sous la forme d'ovales et les acteurs sous la forme de personnages. Il indique également les associations qui les lient.

Exemple :

Le cas d'utilisation de l'achat d'un cheval est représenté par la figure ci-dessous.



Il est possible de représenter le système qui répond au cas d'utilisation sous la forme d'un rectangle englobant le cas.

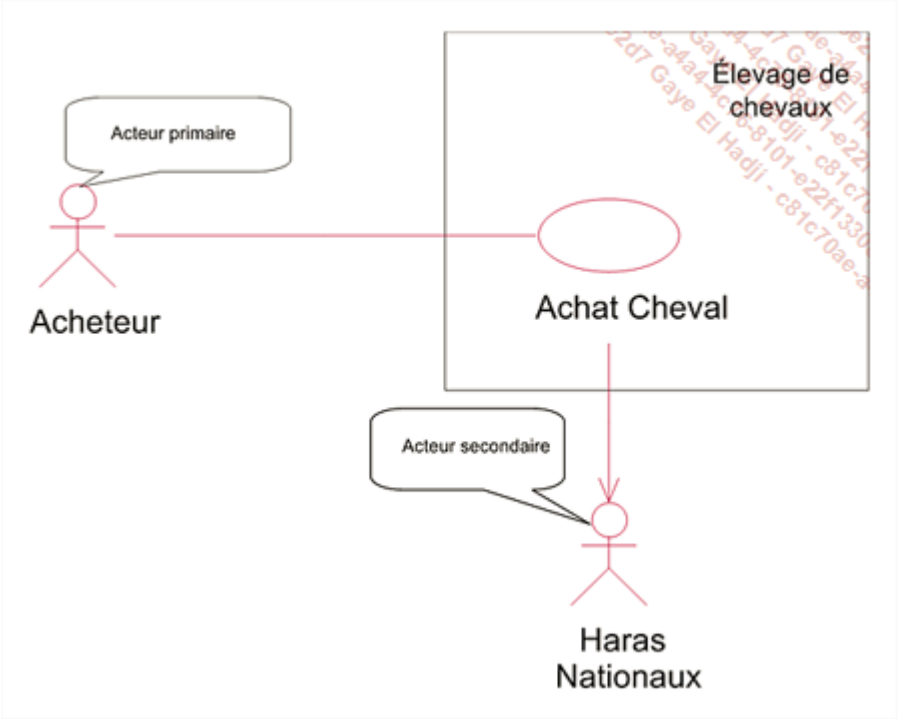
Exemple :

Dans l'exemple précédent, le système est l'élevage de chevaux. Il est illustré à la figure ci-dessous.

Un acteur secondaire est représenté comme un acteur primaire. À la différence de l'association entre un acteur primaire et un cas d'utilisation, l'association entre un acteur secondaire et un cas d'utilisation possède obligatoirement un sens qui va du cas d'utilisation vers l'acteur.

Exemple :

Le changement de propriétaire du cheval est enregistré par les haras nationaux. Ces derniers constituent un acteur secondaire (voir figure ci-dessous).


Élevage de chevaux
Acteur primaire
Acheteur
Achat Cheval
Acteur secondaire
Haras Nationaux

6. *Les cardinalités de l'association acteur/cas d'utilisation*

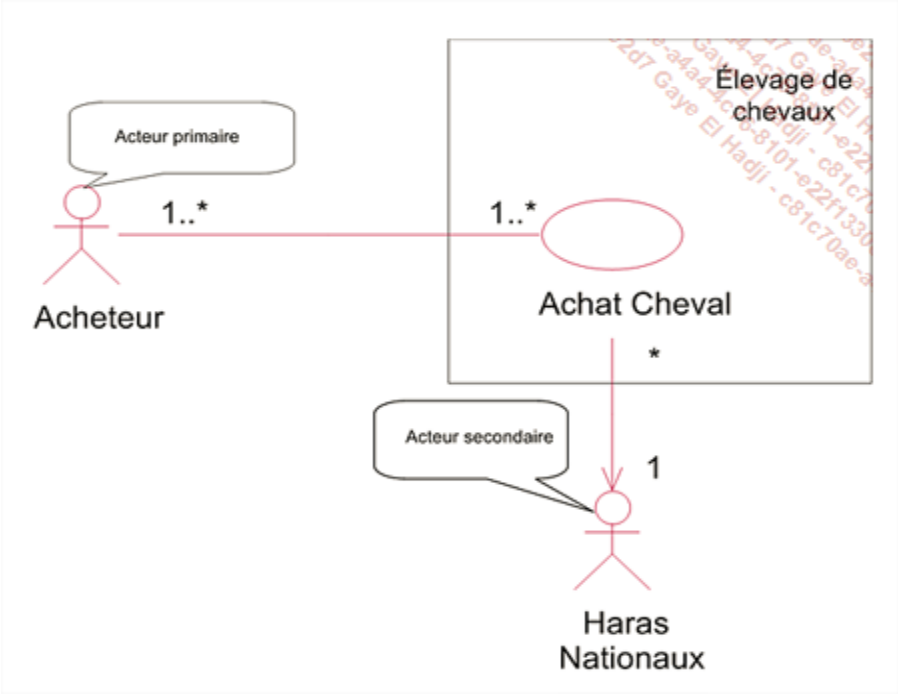
UML offre la possibilité d'introduire des cardinalités au niveau de l'association entre un acteur et un cas d'utilisation. Ces cardinalités figurent au niveau de chaque extrémité de l'association.

La cardinalité située à l'extrémité du cas d'utilisation indique avec combien d'instances du cas d'utilisation (scénarios), chaque instance de l'acteur situé à l'autre extrémité est liée. La cardinalité située à l'extrémité de l'acteur indique avec combien d'instances de l'acteur, chaque instance du cas d'utilisation situé à l'autre extrémité est liée.
Il est possible de spécifier la cardinalité minimale et la cardinalité maximale pour indiquer un intervalle de valeurs auquel doit toujours appartenir la cardinalité. Ces valeurs minimale et maximale sont indiquées dans le tableau ci-après.

Spécification	Cardinalités
0..1	zéro ou une fois
1 = 1,1	une et une seule fois
*	de zéro à plusieurs fois
1..*	de une à plusieurs fois
M..N	entre M et N fois
N	N fois

Exemple :

Le cas d'utilisation de l'achat d'un cheval incluant les cardinalités est représenté par la figure ci-dessous. Un même acheteur peut acheter plusieurs chevaux et peut les acheter en indivision avec d'autres acheteurs. Chaque achat de cheval est enregistré par les haras nationaux qui sont représentés par une seule instance. Ceux-ci enregistrent tous les certificats de vente des chevaux, d'où la présence de la cardinalité * à l'extrémité du cas d'utilisation Achat Cheval.



7. Les relations entre les cas d'utilisation

La relation d'inclusion

La relation d'inclusion sert à enrichir un cas d'utilisation par un autre cas d'utilisation. Cet enrichissement est réalisé par une inclusion impérative, il est donc systématique.

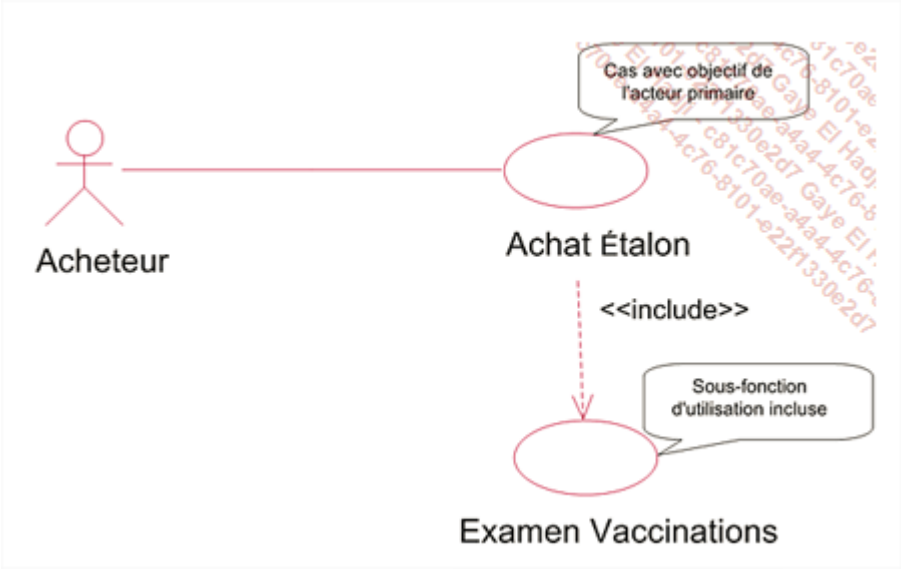
Le cas d'utilisation inclus existe uniquement dans ce but. En effet, il ne répond pas à un objectif d'un acteur primaire. Un tel cas d'utilisation est une sous-fonction.

L'inclusion sert à partager une fonctionnalité commune entre plusieurs cas d'utilisation. Elle peut également être employée pour structurer un cas d'utilisation en décrivant ses sous-fonctions.

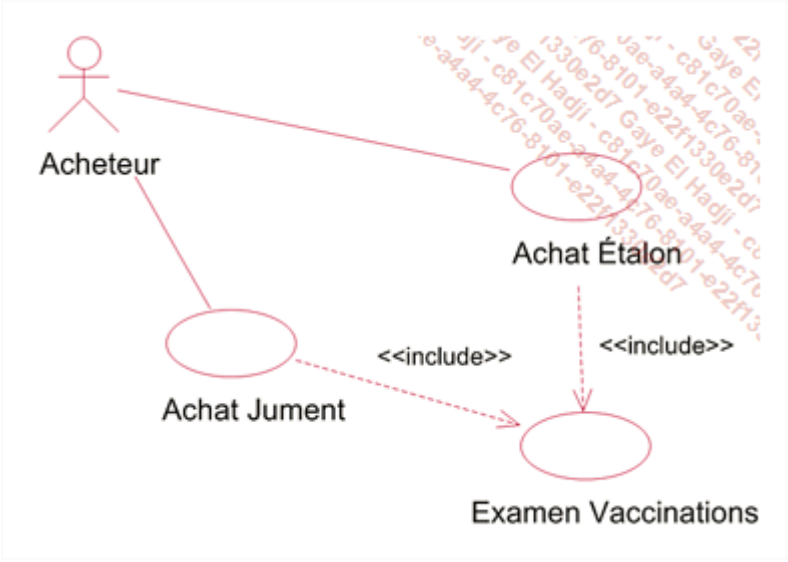
Dans le diagramme des cas d'utilisation, cette relation est représentée par une flèche pointillée munie du stéréotype «include».

Exemple :

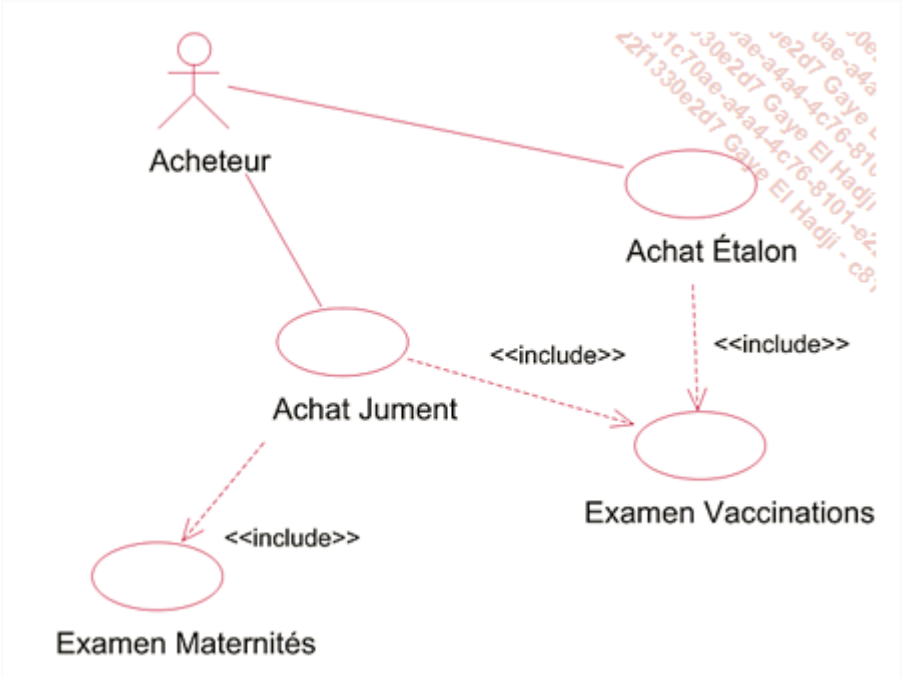
Lors de l'achat d'un étalon, un acheteur vérifie ses vaccinations. Par conséquent, le cas d'utilisation d'achat d'un étalon inclut cette vérification (voir figure ci-dessous).



La mise en commun du cas d'utilisation d'examen des vaccinations est illustrée à la figure ci-dessous car ce cas de sous-fonction est également pertinent pour l'achat d'une jument.



L'inclusion peut également être employée pour décomposer l'intérieur d'un cas d'utilisation sans que le cas inclus soit partagé. À la figure ci-dessous, l'examen des maternités d'une jument n'est pas partagé, mais sa présence illustre bien que cet examen fait partie des points étudiés lors de l'achat d'une jument.



La relation d'extension

Comme la relation d'inclusion, la relation d'extension enrichit un cas d'utilisation par un cas d'utilisation de sous-fonction. Cet enrichissement est analogue à celui de la relation d'inclusion, mais il est optionnel.

L'extension se fait dans le cas d'utilisation de base, en des points précis et prévus lors de la conception, appelés points d'extension.

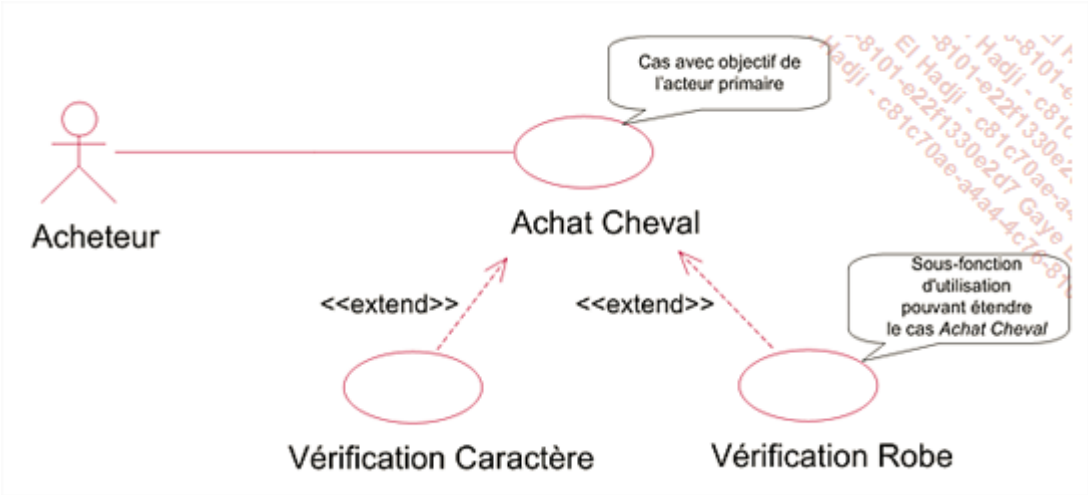
L'application de chaque extension est décidée lors du déroulement d'un scénario. Par conséquent, le cas d'utilisation de base peut être employé sans être étendu.

Comme pour l'inclusion, l'extension sert à structurer un cas d'utilisation ou à partager un cas d'utilisation de sous-fonction.

Dans le diagramme des cas d'utilisation, cette relation est représentée par une flèche pointillée munie du stéréotype «extend».

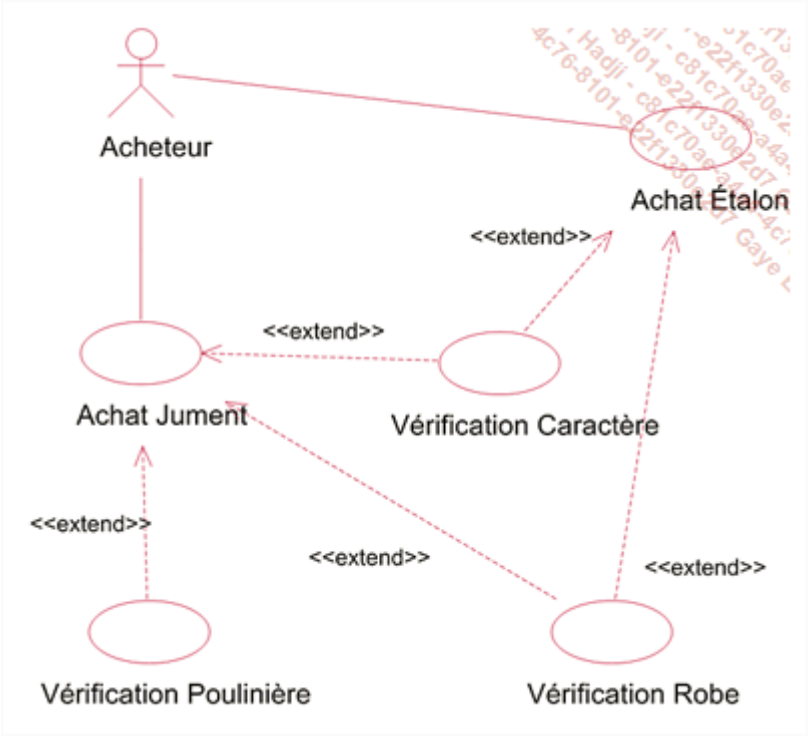
Exemple :

Lors de l'achat d'un cheval, un acheteur peut vérifier son caractère ou sa robe. Par conséquent, le cas d'utilisation d'achat d'un cheval peut être étendu par l'une de ces vérifications (voir figure ci-dessous).



Exemple :

Prenons le cas où l'achat d'un étalon est modélisé séparément de celui d'une jument. Sa capacité à donner naissance peut être vérifiée de façon optionnelle (voir figure ci-dessous). D'autre part, les cas d'utilisation de la vérification du caractère et de la vérification de la robe sont partagés.


Acheteur
Achat Étalon
<<extend>>
<<extend>>
Achat Jument
Vérification Caractère
<<extend>>
<<extend>>
<<extend>>
Vérification Poulinière
Vérification Robe

8. *Tableau des cas d'utilisation*

Il est aussi possible de lister toute les fonctionnalités d'une application en précisant à chaque fois l'acteur et les différents rôles.

Acteur	Rôles
Acteur 1	**Titre fonctionnalité 1**
	1. fonctionnalité : - fonction 1 - fonction 2 - fonction 3
	2. fonctionnalité : - fonction 1 - fonction 2 - fonction 3
	3. fonctionnalité : - fonction 1 - fonction 2 - fonction 3
	Titre fonctionnalité 2
	1. fonctionnalité : - fonction 1 - fonction 2 - fonction 3
	2. fonctionnalité : - fonction 1 - fonction 2 - fonction 3
	3. fonctionnalité : - fonction 1 - fonction 2 - fonction 3

Exemple :

Acteur	Rôles
Acheteur	**Achat cheval**
	1. Récupérer la liste des chevaux:
	2. Identification le cheval pour achat : - identifier un cheval
	3. Papier du cheval : - demander les papier du cheval
	4. Proposition Prix : - faire une proposition d'achat du cheval
	5. Validation vente : - valider la vente du cheval
	6. Papier vente : - Fournir les papiers de vente du cheval
	Vacciner cheval
	1. Vacciner cheval : - Examiner cheval - Doser le vaccin - Vacciner le cheval

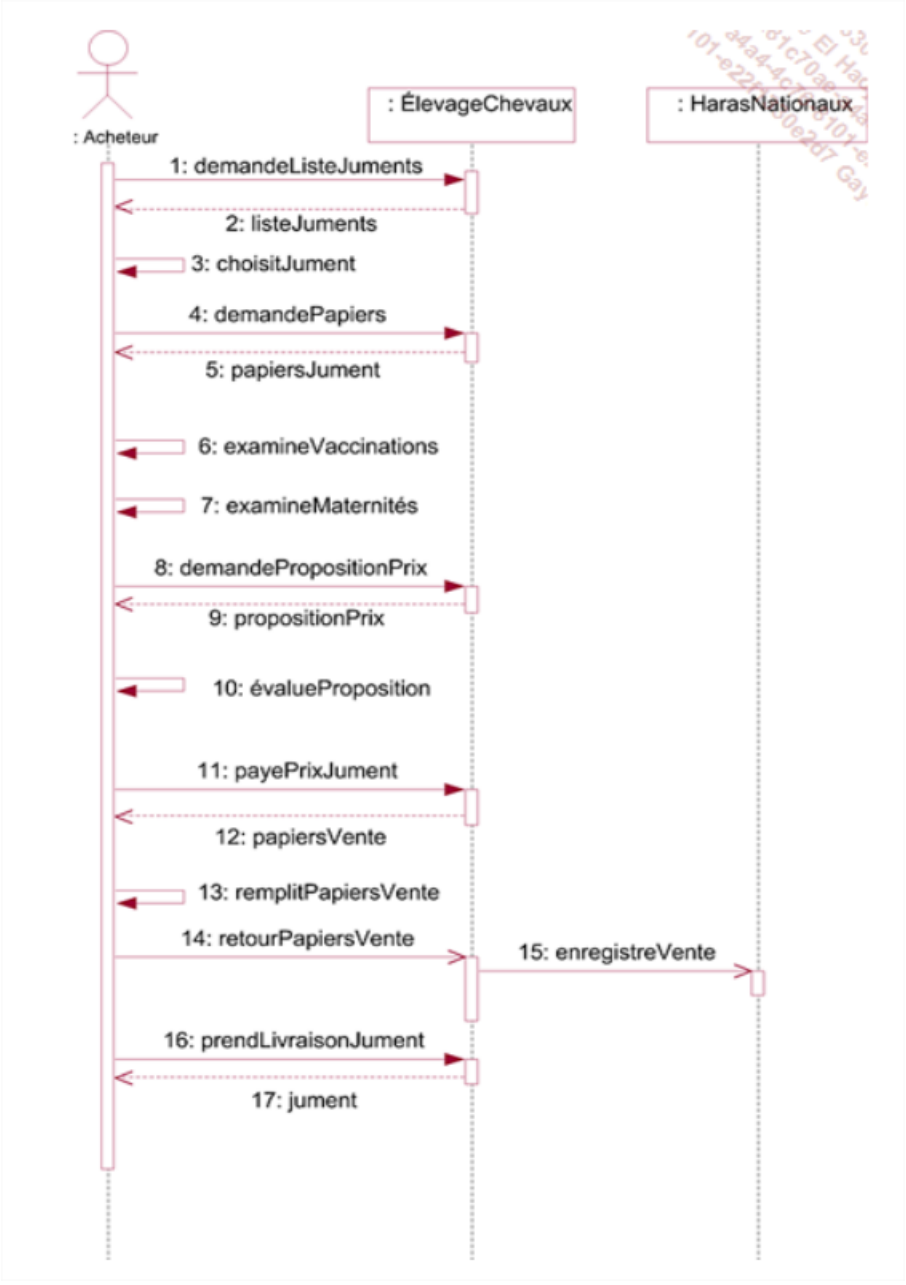
	Enregistrer un cheval

Haras Nationaux	**1. Enregistrement du cheval :** - enregistrer le cheval dans un Haras national

VI) Représentation dynamique en séquences

Nous allons dans cette partie du cours vous faire découvrir les diagrammes de séquence.
Le diagramme de séquence décrit la dynamique du système. À moins de modéliser un très petit système, il est difficile de représenter toute la dynamique d'un système sur un seul diagramme. Aussi la dynamique globale sera représentée par un ensemble de diagrammes de séquence, chacun étant généralement lié à une sous-fonction du système.

Le diagramme de séquence décrit les interactions entre un groupe d'objets en montrant, de façon séquentielle, les envois de message qui interviennent entre les objets. Le diagramme peut également montrer les transmissions de données échangées lors des envois de message.



VII) Les concepts de POO

Cette partie du cours a pour objectif de vous faire découvrir les différents concepts et principes de l'approche objet qui sont à la base d'UML. Leur connaissance est indispensable pour comprendre les éléments utilisés dans la panoplie des diagrammes d'UML qui seront abordés dans les chapitres suivants.

Dans un premier temps, nous examinerons le concept d'objet, puis nous verrons comment le modéliser, par abstraction, en UML.

La notion de classes, représentation commune d'un ensemble d'objets similaires, sera introduite.

Nous évoquerons ensuite le principe d'encapsulation, masquage d'informations internes et propres au fonctionnement de l'objet.

Les relations de spécialisation et de généralisation introduisant les hiérarchies de classes seront décrites, ainsi que l'héritage, les classes concrètes et abstraites, puis nous aborderons le polymorphisme, conséquence directe de la spécialisation.

Enfin, nous évoquerons la composition d'objets avant de terminer sur une notion plus spécifique à UML, la spécialisation des éléments du diagramme par les stéréotypes.

1. *L'objet*

Un objet est une entité identifiable du monde réel. Il peut avoir une existence physique (un cheval, un livre) ou ne pas en avoir (un texte de loi). Identifiable signifie que l'objet peut être désigné.

Exemple :

Ma jument Jorphée
Mon livre sur UML
L'article 293B du code des impôts

En UML, tout objet possède un ensemble d'attributs (sa structure) et un ensemble de méthodes (son comportement). Un attribut est une variable destinée à recevoir une valeur. Une méthode est un ensemble d'instructions prenant des valeurs en entrée et modifiant les valeurs des attributs ou produisant un résultat.

Même un objet statique du monde réel est toujours perçu comme dynamique. Ainsi en UML, un livre est perçu comme un objet capable de s'ouvrir lui-même à la énième page.

Tout système conçu en UML est composé d'objets interagissant entre eux et effectuant les opérations propres à leur comportement.

Exemple :

Un troupeau de chevaux est un système d'objets interagissant entre eux, chaque objet possédant son propre comportement.

Le comportement global d'un système est ainsi réparti entre les différents objets. Dans notre exemple, il suffit de faire le parallèle avec le monde réel pour le comprendre.

2. *L'abstraction*

L'abstraction est un principe très important en modélisation. Elle consiste à retenir uniquement les propriétés pertinentes d'un objet pour un problème précis. Les objets utilisés en UML sont des abstractions d'objets du monde réel.

Exemple :

On s'intéresse aux chevaux pour l'activité de course. Les propriétés d'aptitude de vitesse, d'âge et d'équilibre mental ainsi que l'élevage d'origine sont pertinentes pour cette activité et sont retenues.

On s'intéresse aux chevaux pour l'activité de trait. Les propriétés d'âge, de taille, de force et de corpulence sont pertinentes pour cette activité et sont retenues.

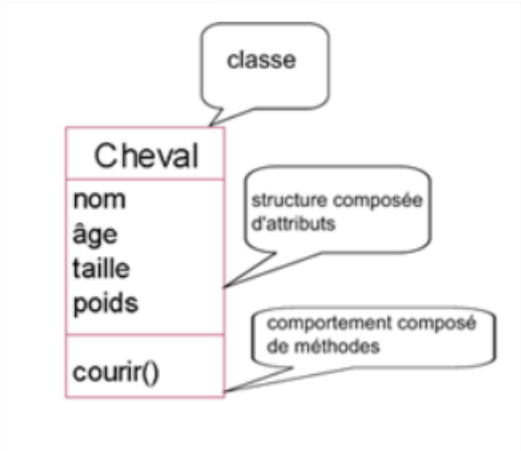
L'abstraction est une simplification indispensable au processus de modélisation. Un objet UML est donc une abstraction de l'objet du monde réel par rapport aux besoins du système, dont on ne retient que les éléments essentiels.

3. *Les classes d'objets*

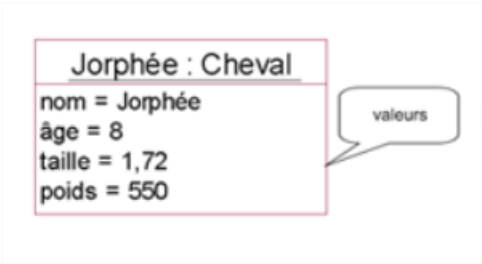
Un ensemble d'objets similaires, c'est-à-dire possédant la même structure et le même comportement et constitués des mêmes attributs et méthodes, forme une classe d'objets. La structure et le comportement peuvent alors être définis en commun au niveau de la classe.

Chaque objet d'une classe, encore appelé instance de classe, se distingue par son identité propre et possède des valeurs spécifiques pour ses attributs.

Exemple : L'ensemble des chevaux constitue la classe Cheval qui possède la structure et le comportement décrits à la figure ci-dessous.



Le cheval Jorphée est une instance de la classe Cheval dont les attributs et leurs valeurs sont illustrés à la figure ci-dessous.



Le nom d'une classe apparaît au singulier. Il est constitué d'un nom commun précédé ou suivi d'un ou plusieurs adjectifs qualifiant le nom. Ce nom est significatif de l'ensemble des objets constituant la classe.

4. *L'encapsulation*

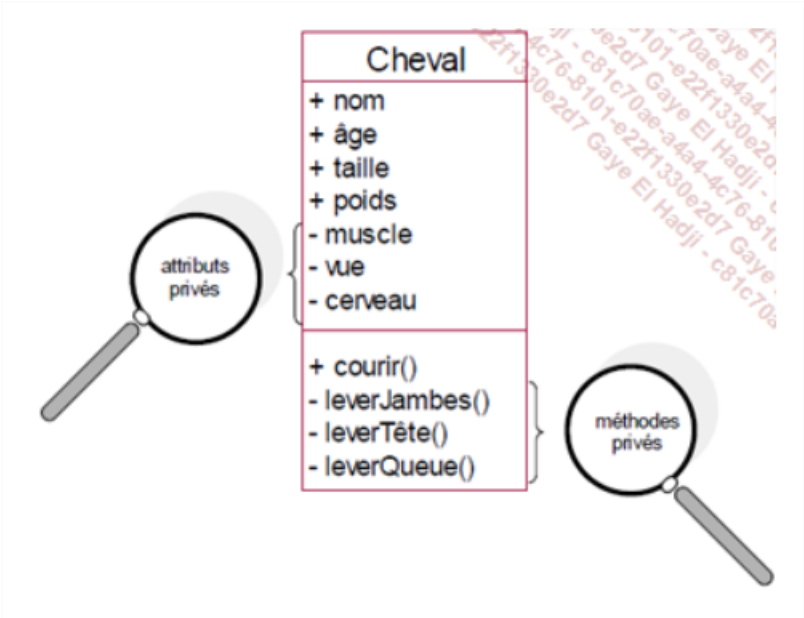
L'encapsulation consiste à masquer des attributs et des méthodes de l'objet vis-à-vis des autres objets. En effet, certains attributs et méthodes ont pour seul objectif des traitements internes à l'objet et ne doivent pas être exposés aux objets extérieurs. Encapsulés, ils sont appelés les attributs et méthodes privés de l'objet.

L'encapsulation est une abstraction puisque l'on simplifie la représentation de l'objet vis-à-vis des objets extérieurs. Cette représentation simplifiée est constituée des attributs et méthodes publics de l'objet.

La définition de l'encapsulation se fait au niveau de la classe. Les objets extérieurs à un objet sont donc les instances des autres classes.

Exemple

Lorsqu'il court, un cheval effectue différents mouvements comme lever les jambes, lever la tête, lever la queue. Ces mouvements sont internes au fonctionnement de l'animal et n'ont pas à être connus à l'extérieur. Ce sont des méthodes privées. Ces opérations accèdent à une partie interne du cheval : ses muscles, son cerveau et sa vue. Cette partie interne est représentée sous la forme d'attributs privés. L'ensemble de ces attributs et méthodes est illustré à la figure ci-dessous.



Dans la notation UML, les attributs et méthodes publics sont précédés du signe plus, tandis que les attributs et méthodes privés (encapsulés) sont précédés du signe moins.

5. *La spécialisation et la généralisation*

Une classe d'objets peut être introduite séparément des autres classes ou bien définie comme un sous-ensemble d'une autre classe, ce sous-ensemble devant toujours constituer un ensemble d'objets similaires.

Il s'agit alors d'une sous-classe d'une autre classe. Elle constitue ainsi une spécialisation de cette autre classe.

Exemple :

La classe des chevaux est une sous-classe de la classe des mammifères.

La généralisation est la relation inverse de la spécialisation. Si une classe Cheval est une spécialisation d'une classe Mammifères, Cheval est une généralisation de Mammifères. Mammifères est la surclasse de Cheval.

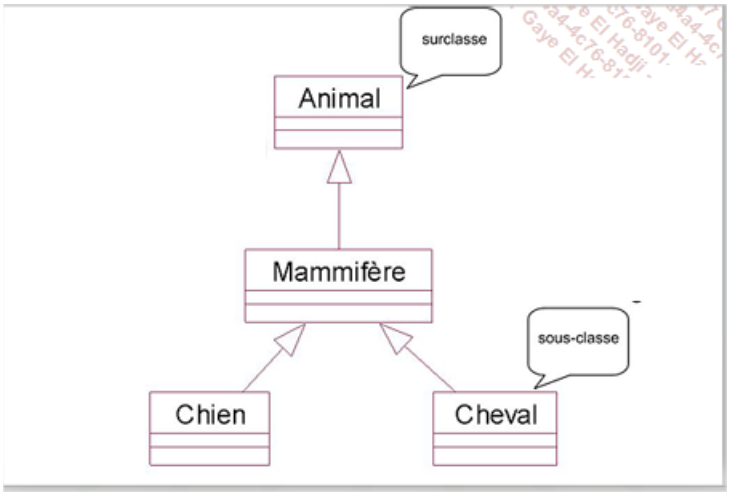
Exemple :

La classe des mammifères est une surclasse de la classe des chevaux.

La relation de spécialisation peut s'appliquer à plusieurs niveaux, donnant lieu à une hiérarchie de classes.

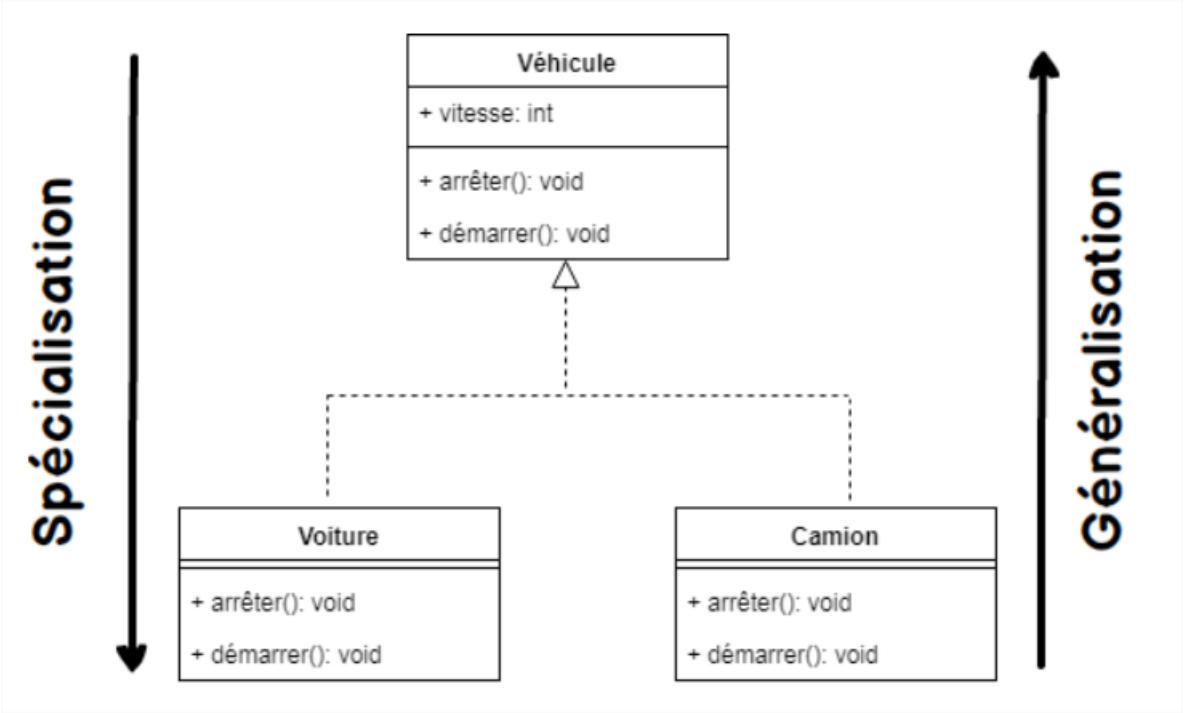
Exemple :

La classe des chevaux est une sous-classe de la classe des mammifères, elle-même sous-classe de la classe des animaux. La classe des chiens est une autre sous-classe de la classe des mammifères. La hiérarchie correspondante des classes est représentée à la figure ci-dessous.



Exemple :

La classe des voitures est une sous-classe de la classe des véhicules. La hiérarchie correspondante des classes est représentée à la figure ci-dessous.

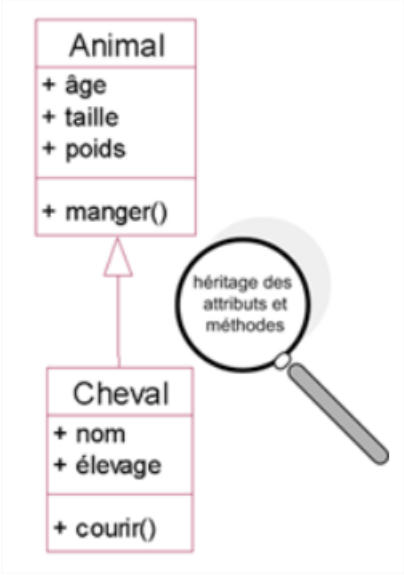


6. *L'héritage*

L'héritage est la propriété qui fait bénéficier à une sous-classe de la structure et du comportement de sa surclasse. L'héritage provient du fait qu'une sous-classe est un sous-ensemble de sa surclasse. Ses instances sont également instances de sa surclasse. En conséquence, elles bénéficient de la structure et du comportement définis dans cette surclasse, en plus de la structure et du comportement introduits au niveau de la sous-classe.

Exemple :

Soit un système où la classe Cheval est une sous-classe directe de la classe Animal, un cheval est alors décrit par la combinaison de la structure et du comportement issus des classes Cheval et Animal, c'est-à-dire avec les attributs âge, taille, poids, nom et élevage ainsi que les méthodes manger et courir. Cet héritage est illustré à la figure ci-dessous.



L'héritage est une conséquence de la spécialisation. Cependant, les informaticiens emploient beaucoup plus souvent le terme hérite que spécialise pour désigner la relation entre une sous-classe et sa surclasse.

7. *Les classes abstraites et concrètes*

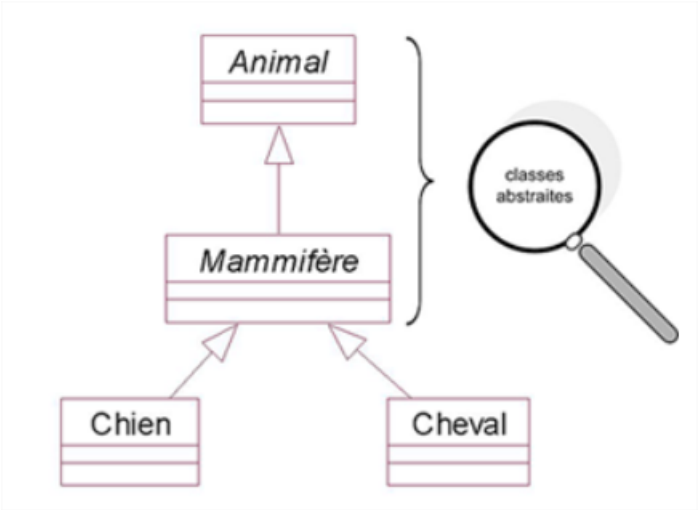
L'examen de la hiérarchie présentée à la figure précédente montre qu'il existe deux types de classes dans la hiérarchie :

- Des classes qui possèdent des instances, à savoir les classes Cheval et Chien. Ces classes sont appelées classes concrètes.
- Des classes qui n'en possèdent pas directement, comme la classe Animal. En effet, si dans le monde réel, il existe des chevaux, des chiens, le concept d'animal reste, quant à lui, abstrait. Il ne suffit pas à définir complètement un animal. La classe Animal est appelée une classe abstraite.

Une classe abstraite a pour vocation de posséder des sous-classes concrètes. Elle sert à factoriser des attributs et des méthodes communs à ses sous-classes.

Exemple :

La figure ci-dessous reprend la hiérarchie en indiquant précisément les classes abstraites et les classes concrètes. En UML, le nom des classes abstraites apparaît en caractères italiques.



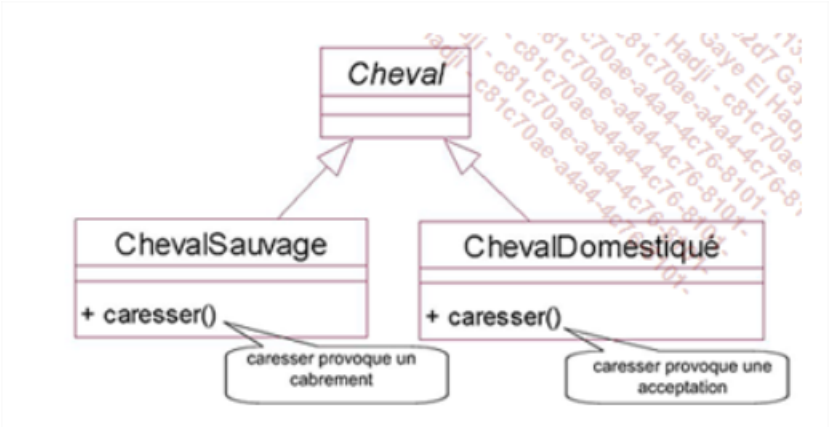
8. *Le polymorphisme*

Le polymorphisme signifie qu'une classe (très généralement abstraite) représente un ensemble constitué d'objets différents car ils sont instances de sous-classes distinctes. Lors de l'appel d'une méthode de même nom, cette différence se traduit par des comportements différents (sauf dans le cas où la méthode est commune et héritée de la surclasse dans les sous-classes).

Exemple :

Soit la hiérarchie de classes illustrée à la figure ci-dessous. La méthode « **caresser** » a un comportement différent selon que le cheval est instance de **ChevalSauvage** ou de **ChevalDomestiqué**. Dans le premier cas, le comportement sera un refus (se traduisant par un cabrement) alors que, dans le second, le comportement sera une acceptation.

Si l'on considère la classe Cheval dans son intégralité, on a donc un ensemble de chevaux qui ne réagissent pas de la même façon lors de l'activation de la méthode **caresser**.

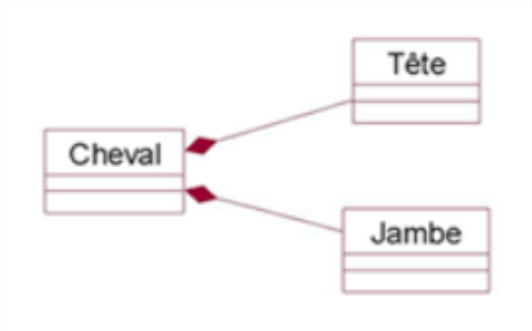


9. *La composition*

Un objet peut être complexe et composé d'autres objets. L'association qui unit alors ces objets est la composition. Elle se définit au niveau de leurs classes, mais les liens sont bâtis entre les instances des classes. Les objets formant l'objet composé sont appelés composants.

Exemple :

Un cheval est un exemple d'objet complexe. Il est constitué de ses différents organes (jambes, tête, etc.). La représentation graphique de cette composition se trouve à la figure ci-dessous.



La composition peut prendre deux formes :

- la composition faible ou agrégation ;
- la composition forte.

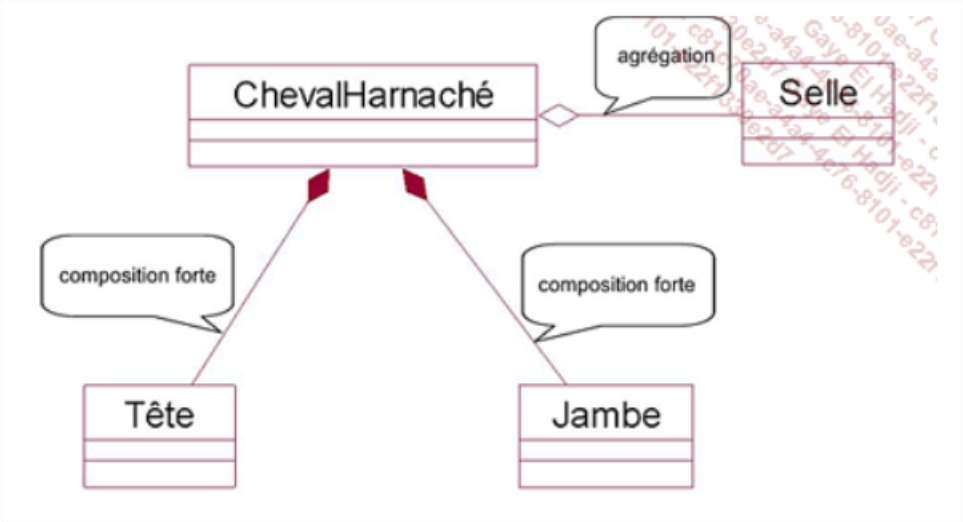
Dans la composition faible, les composants peuvent être partagés entre plusieurs objets complexes. Dans la composition forte, les composants ne peuvent être partagés et la destruction de l'objet composé entraîne la destruction de ses composants.

Exemple :

Si l'on reprend l'exemple précédent dans le cas d'un cheval de course harnaché et si l'on ajoute la selle dans les composants, on obtient :

- Une composition forte pour les jambes et la tête ; en effet, jambes et tête ne peuvent pas être partagées et la disparition du cheval entraîne la disparition de ses organes.
- Une agrégation ou composition faible pour la selle.

L’ensemble est illustré à la figure ci-dessous.



10. La spécialisation des éléments : la notion de stéréotype en UML

Nous avons introduit dans ce chapitre les concepts de l'approche par objets. Nous introduisons maintenant les stéréotypes d'UML dont le but est de spécialiser ces concepts.

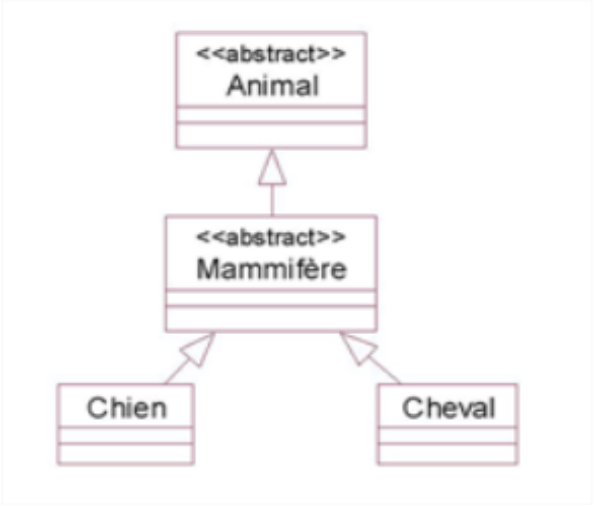
Un stéréotype est constitué d'un mot-clé explicitant cette spécialisation. Celui-ci est noté entre guillemets.

Cette spécialisation est réalisée indépendamment du système que l'on cherche à modéliser.

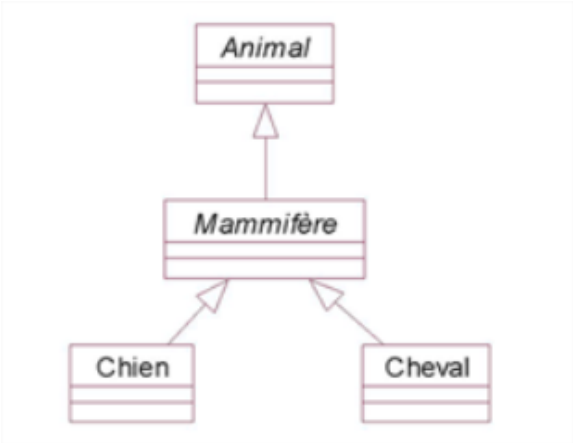
Exemple :

Le concept de classe abstraite est un concept spécialisé du concept de classe. Nous avons vu qu'une classe abstraite est représentée comme une classe avec un nom en italique. Cette représentation graphique inclut un stéréotype implicite, mais il est également possible de ne pas mettre le nom de la classe en italique et de préciser explicitement le stéréotype «abstract».

Ce stéréotype explicite est illustré à la figure ci-dessous. Il peut être employé lors de l'écriture manuelle de diagrammes UML.



équivalent à :



11. Conclusion

L'approche par objets forme la base d'UML. Elle est constituée de concepts (objets, classes, spécialisation, composition) et de principes (abstraction, encapsulation). Cet ensemble fait de l'approche par objets un véritable support pour la modélisation de systèmes complexes, et au-delà d'UML, pour leur programmation.

Nous verrons dans les chapitres suivants comment les différents diagrammes d'UML s'appuient sur les concepts et principes de l'approche par objets.

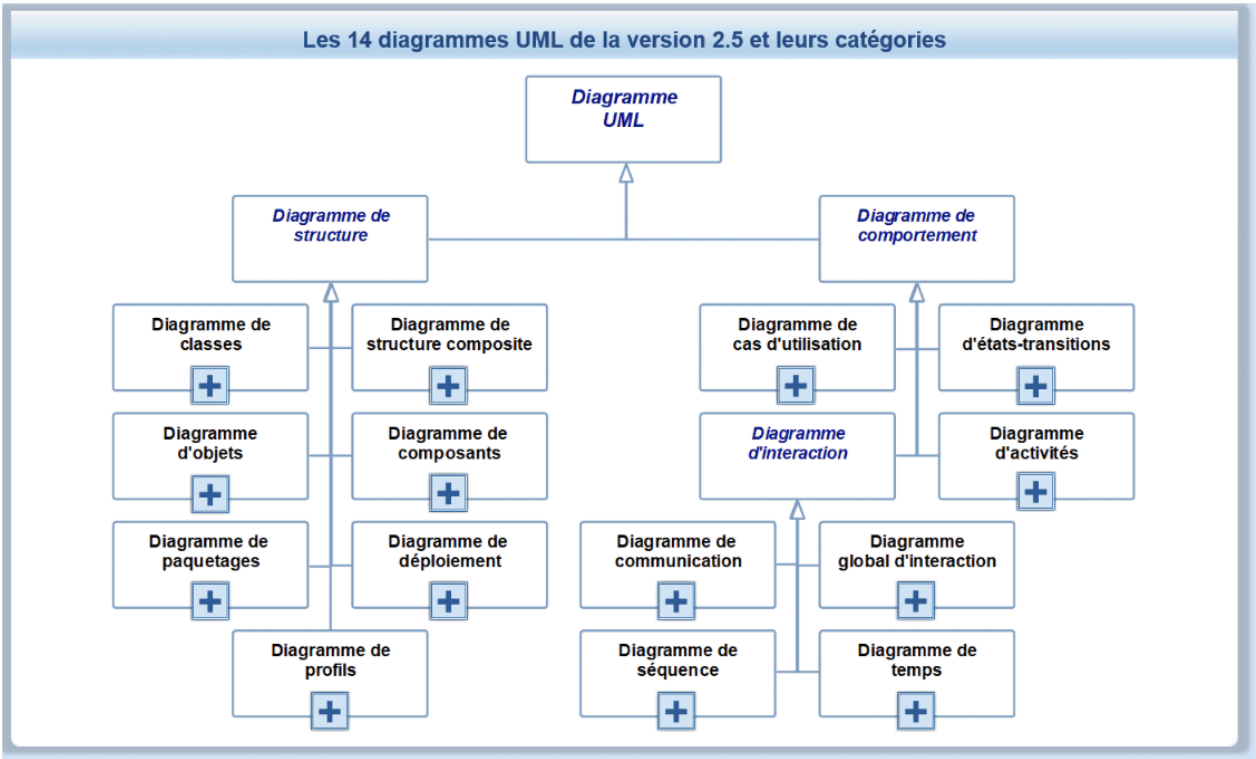
VIII) Les différents types diagrammes d'UML

La **version 2.5.1** du language UML publié par OMG (Object Management Group) en décembre 2017, definit et spécifie 14 diagramme standards.

- 7 diagrammes qui servent à décrire les aspects structurels d'un système informatique (logiciel et matériel).

- 7 diagrammes qui servent à décrire les aspects dynamiques (comportement) du système.

Voici ci-dessous un schéma illustrant les 14 diagrammes :



Il faut noter que **9 diagrammes** dans la première version UML. D'autres diagrammes sont été ajoutés par la suite. Vous trouverez ci-dessous un tableau retraçant toute l'historique d'ajout de nouveaux diagrammes.

Diagramme de…	Version et année																	
	0.8	0.9	0.9.1	1.0	1.1	1.2	1.3	1.4	1.5	2.0	2.1.1	2.1.2	2.2	2.3	2.4	2.4.1	2.5	2.5.1
	1995	1996	1996	1997	1997	1998	2000	2001	2003	2005	2007	2007	2009	2010	2011	2011	2015	2017
Structure (*Structure*)																		
Classes (*Class*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Objets (*Object*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Paquetages (*Package*)										x	x	x	x	x	x	x	x	x
Profils (*Profile*)													x	x	x	x	x	x
Déploiement (*Deployment*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Composants (*Component*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Structure composite (*Composite Structure*)										x	x	x	x	x	x	x	x	x
Comportement (*Behavior*)																		
Cas d'utilisation (*Use Case*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
États-transitions (*State Machine*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Activités (*Activity*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Global d'interaction (*Interaction Overview*)										x	x	x	x	x	x	x	x	x
Temps (*Timing*)										x	x	x	x	x	x	x	x	x
Séquence (*Sequence*)	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Collaboration (*Collaboration*)	x	x	x	x	x	x	x	x	x									
Communication (*Communication*)										x	x	x	x	x	x	x	x	x
Nombre de diagrammes par version	9	9	9	9	9	9	9	9	9	13	13	13	14	14	14	14	14	14

1. *Diagramme de cas d'utilisation*

Le diagramme des cas d'utilisation montre les cas d'utilisation représentés sous la forme d'ovales et les acteurs sous la forme de personnages. Il indique également les associations qui les lient.

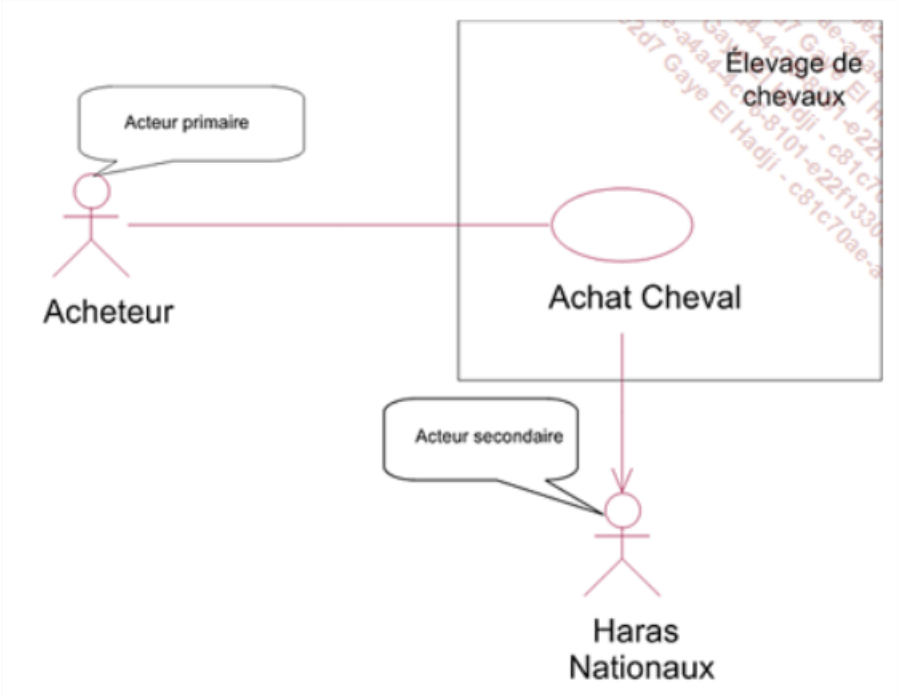
Exemple : Le cas d'utilisation de l'achat d'un cheval est représenté par la figure ci-dessous :

Il est possible de représenter le système qui répond au cas d'utilisation sous la forme d'un rectangle englobant le cas.

Exemple : Dans l'exemple précédent, le système est l'élevage de chevaux. Il est illustré à la figure ci-dessous.

Un acteur secondaire est représenté comme un acteur primaire. À la différence de l'association entre un acteur primaire et un cas d'utilisation, l'association entre un acteur secondaire et un cas d'utilisation possède obligatoirement un sens qui va du cas d'utilisation vers l'acteur.

Exemple : Le changement de propriétaire du cheval est enregistré par les haras nationaux. Ces derniers constituent un acteur secondaire (voir figure ci-dessous).



2. *Diagramme de classes*

L'objectif de cette partie du cours est de vous faire découvrir les techniques UML de modélisation statique des objets.

Cette modélisation est statique car elle ne décrit pas les interactions ou le cycle de vie des objets. Les méthodes sont introduites d'un point de vue statique, sans description de leur enchaînement.

Nous découvrirons le diagramme de classes. Ce diagramme contient les attributs, les méthodes et les associations des objets.

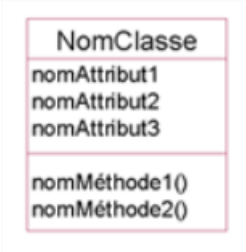
Ce diagramme est central lors d'une modélisation par objets d'un système. De tous les diagrammes UML, il est le seul obligatoire lors d'une telle modélisation.

Nous étudierons comment le langage **OCL** (Object Constraint Language ou langage de contraintes objet) peut étendre le diagramme de classes pour exprimer de façon plus riche les contraintes. Ensuite, le diagramme d'objets nous montrera comment illustrer la modélisation réalisée dans le diagramme de classes.

L'emploi d'OCL, du diagramme d'objets ou du diagramme de structure composite est optionnel. Leur utilisation dépend des contraintes du projet de modélisation.

Représentation des classes

Les objets du système sont décrits par des classes dont une forme simplifiée de la représentation en UML est donnée à la figure suivante. Cette représentation est constituée de trois parties.



La première partie contient le nom de la classe.

Remarque : Rappelons que le nom d'une classe est au singulier. Il est constitué d'un nom commun précédé ou suivi d'un ou plusieurs adjectifs qualifiant le nom. Ce nom est significatif de l'ensemble des objets constituant la classe. Il représente la nature des instances d'une classe.

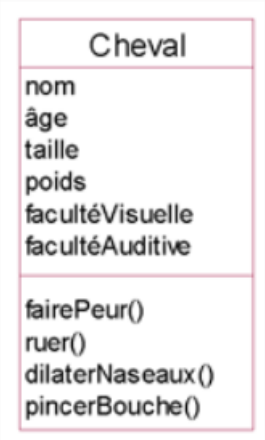
La deuxième partie contient les attributs. Ceux-ci contiennent l'information portée par un objet. L'ensemble des attributs forme la structure de l'objet.

La troisième partie contient les méthodes. Celles-ci correspondent aux services offerts par l'objet. Elles peuvent modifier la valeur des attributs. L'ensemble des méthodes forme le comportement de l'objet.

Remarque : Le nombre d'attributs et de méthodes est variable selon chaque classe. Toutefois, un nombre élevé d'attributs et/ou de méthodes est déconseillé. Il ne reflète pas, en général, une bonne conception de la classe.

Exemple :

La figure suivante montre la classe Cheval comme exemple de la représentation simplifiée d'une classe en UML.



Cette forme de représentation des classes est la plus simple car elle ne fait pas apparaître les caractéristiques des attributs et des méthodes en dehors de leur nom. Elle est très souvent utilisée lors d'une première phase de modélisation.

Avant d'examiner les représentations plus complètes, nous devons aborder les notions essentielles d'encapsulation, de type et de signature des méthodes.

L'encapsulation a été introduite précédemment. Certains attributs et méthodes ne sont pas exposés à l'extérieur de l'objet. Ils sont encapsulés et appelés attributs et méthodes privés de l'objet.

UML, comme la plupart des langages à objets modernes, introduit trois possibilités d'encapsulation :

- L'attribut privé ou la méthode privée : la propriété n'est pas exposée en dehors de la classe, y compris au sein de ses sous-classes.
- L'attribut protégé ou la méthode protégée : la propriété n'est exposée qu'aux instances de la classe et de ses sous-classes.
- L'encapsulation de paquetage : la propriété n'est exposée qu'aux instances des classes de même paquetage.

La notion de propriété privée est rarement utilisée car elle amène à instaurer une différence entre les instances d'une classe et les instances de ses sous-classes. Cette différence est liée à des aspects assez subtils de la programmation par objets. Quant à l'encapsulation de paquetage, elle provient du langage Java. Elle est réservée à l'écriture de diagrammes destinés aux développeurs.

Nous suggérons l'utilisation de l'encapsulation protégée.

L'encapsulation est représentée par un signe plus, un signe moins, un dièse, ou un tilde précédant le nom de l'attribut. Le tableau suivant détaille la signification de ces signes.

public	+	Élément non encapsulé visible par tous.
protégé	#	Élément encapsulé visible dans les sous-classes de la classe.
privé	-	Élément encapsulé visible seulement dans la classe.
paquetage	~	Élément encapsulé visible seulement dans les classes du même paquetage.

Exemple

La figure suivante montre la classe Cheval en faisant ressortir les caractéristiques d'encapsulation.

Nous appelons ici variable tout attribut, paramètre et valeur de retour d'une méthode. D'une façon générale, nous appelons variable tout élément pouvant prendre une valeur.

Le type est une contrainte appliquée à une variable. Il consiste à fixer l'ensemble des valeurs possibles que peut prendre cette variable. Cet ensemble peut être une classe, auquel cas la variable doit contenir une référence vers une instance de cette classe. Il peut être standard comme l'ensemble des entiers, des chaînes de caractères, des booléens ou des réels. Dans ces derniers cas, la valeur de la variable doit être respectivement un entier, une chaîne de caractères, une valeur booléenne et un réel.

Les types standards sont désignés ainsi :

- Integer pour le type des entiers ;
- String pour le type des chaînes de caractères ;
- Boolean pour le type des booléens ;
- Real pour le type des réels.

Exemple : 1 ou 3 ou 10 sont des exemples de valeurs d'entier. "Cheval" est un exemple de chaîne de caractères où les guillemets ont été choisis comme séparateurs. False et True sont les deux seules valeurs possibles du type Boolean.

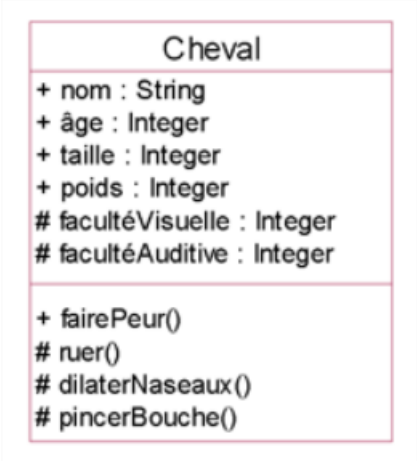
3.1415, où le point a été choisi comme séparateur des décimales, est un exemple bien connu de nombre réel.

Nous verrons que l'on ne fait généralement recours à une classe pour typer un attribut que si cette dernière est une classe d'une bibliothèque externe au système modélisé ou une interface. Il est possible d'utiliser une classe du système pour donner un type à un attribut. Cependant, dans ce cas, il est souvent préférable de recourir aux associations inter objets.

En revanche, le type d'un paramètre ou du retour d'une méthode peut être un type standard ou une classe, qu'elle appartienne ou non au système.

Le type d'un attribut, d'un paramètre et de la valeur de retour d'une méthode est précisé au niveau de la représentation de la classe.

Exemple : La figure suivante montre la classe Cheval pour laquelle le type de tous les attributs a été fixé. Les attributs de cette classe utilisent des types standards.



La cardinalité

Une variable (attribut, paramètre et valeur de retour d'une méthode) peut contenir plusieurs valeurs. Dans la plupart des langages de programmation, une telle variable est appelée un tableau ou une liste.

La cardinalité est indiquée à la suite du type avec la syntaxe suivante :

[borneInf..borneSup]

Le nombre de valeurs que prend la variable est alors compris entre borneInf et borneSup. Il est possible d'indiquer une seule borne qui précise le nombre de valeurs que doit prendre exactement la variable. Le symbole * peut être utilisé comme borneSup. Il signifie qu'il n'y a pas de borne supérieure limitant le nombre de valeurs que peut prendre la variable. La syntaxe [0..*] peut également être écrite [*].

Exemple : Nous faisons l'hypothèse qu'un cheval peut posséder plusieurs noms avec un minimum de 1 et un maximum de 3. La syntaxe pour l'attribut nom est alors la suivante :

+ nom : String[1..3]

Les propriétés des variables

UML permet d'attribuer des propriétés à une variable (attribut, paramètre et valeur de retour d'une méthode) en les indiquant entre accolades. Les propriétés suivantes figurent parmi les plus utilisées.

{readOnly} : cette propriété indique que la valeur de la variable ne peut pas être modifiée. Elle doit être initialisée avec une valeur par défaut.

{redefines nomAttribut} : cette propriété n'est applicable qu'à un attribut. Elle spécifie la redéfinition de l'attribut de nom **nomAttribut** de l'une des surclasses. La redéfinition peut notamment porter sur le nom de l'attribut ou sur son type. En cas de changement de type, le nouveau type doit être compatible avec l'ancien, c'est-à-dire que son ensemble de valeurs doit être inclus dans l'ensemble des valeurs de l'ancien type.

{ordered} : lorsqu'une variable peut contenir plusieurs valeurs (cardinalité supérieure à 1), les valeurs doivent être ordonnées.

{unique} : lorsqu'une variable peut contenir plusieurs valeurs (cardinalité supérieure à 1), chaque valeur doit être unique (interdiction d'avoir des doublons). Cette propriété s'applique par défaut.

{nonunique} : lorsqu'une variable peut contenir plusieurs valeurs (cardinalité supérieure à 1), la présence de doublons est possible.

Pour attribuer plusieurs propriétés à une variable, il faut les séparer par des virgules.

Exemple : Nous reprenons l'hypothèse qu'un cheval peut posséder plusieurs noms avec un minimum de 1 et un maximum de 3. Nous désirons maintenant que l'attribut multivalué soit composé de noms uniques et ordonnés. Nous utilisons alors la syntaxe suivante :

```
+ nom : String[1..3]{unique, ordered}
```

La signature des méthodes

Une méthode d'une classe peut prendre des paramètres et renvoyer un résultat. Les paramètres sont des valeurs transmises :

- À l'aller, lors de l'envoi d'un message appelant la méthode ;
- Ou au retour d'appel de la méthode.

Le résultat est une valeur transmise à l'objet appelant lors du retour d'appel.

Ces paramètres ainsi que le résultat peuvent être typés. L'ensemble constitué du nom de la méthode, des paramètres avec leur nom, leur type, leur cardinalité, leurs propriétés ainsi que du type du résultat avec sa cardinalité et ses propriétés s'appelle la signature de la méthode.

Une signature prend la forme suivante :

nomMéthode (direction nomParamètre :
type[borneInf..borneSup]=ValeurDéfaut{propriétés}, ...) :
typeRésultat[borneInf..borneSup]{propriétés}

Rappelons que le nombre de paramètres peut être nul et que le type du résultat est optionnel.

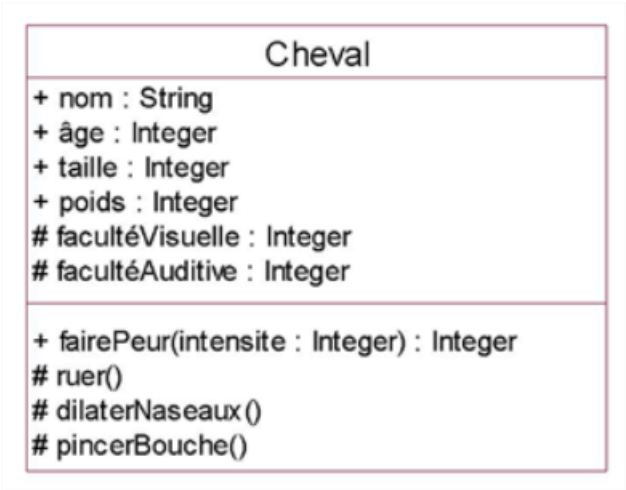
Devant le nom du paramètre, il est possible d'indiquer par un mot-clé la direction dans laquelle celui-ci est transmis. Les trois mots-clés possibles sont :

- in : la valeur du paramètre n'est transmise qu'à l'appel ;
- out : la valeur du paramètre n'est transmise qu'au retour de l'appel de la méthode ;
- inout : la valeur du paramètre est transmise à l'appel et au retour.

Si aucun mot-clé n'est précisé, la valeur du paramètre n'est transmise qu'à l'appel.

Remarque : Les directions out et inout ne sont pas compatibles avec un appel en mode asynchrone où l'appelant n'attend pas le retour d'appel de la méthode.

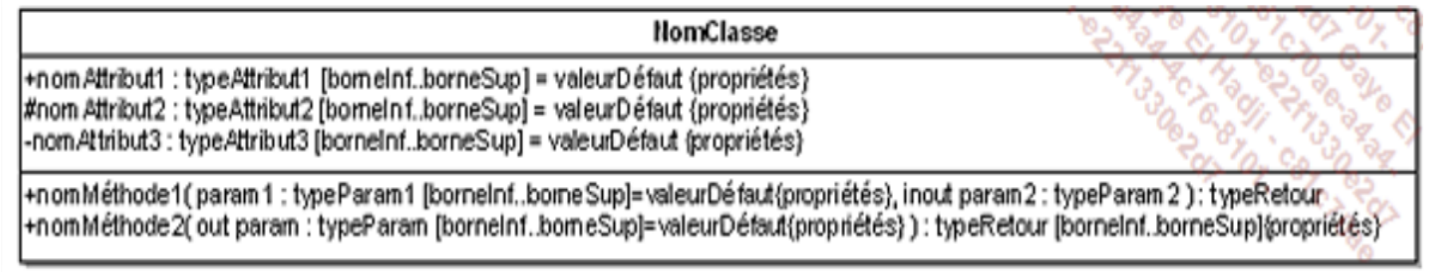
Exemple : La figure suivante montre la classe Cheval dont la méthode **fairePeur** a été munie d'un paramètre, à savoir l'intensité avec laquelle le cavalier fait peur, et d'un retour qui est l'intensité de la peur que le cheval ressent. Ces deux valeurs sont des entiers. Quant aux autres méthodes, elles ne prennent pas de paramètres et ne renvoient pas de résultat.



La représentation complète des classes fait apparaître les attributs avec leur caractéristique d'encapsulation, leur type et les méthodes avec leur signature complète.

Il est également possible de conférer des valeurs par défaut aux attributs et aux paramètres d'une méthode. La valeur par défaut d'un attribut est celle qui lui est attribuée dès la création d'un nouvel objet. La valeur par défaut d'un paramètre est utilisée lorsque l'appelant d'une méthode ne fournit pas explicitement la valeur de ce paramètre au moment de l'appel.

La figure suivante illustre la représentation complète d'une classe. Il est bien sûr possible de choisir une représentation intermédiaire entre la représentation simplifiée et la représentation complète.



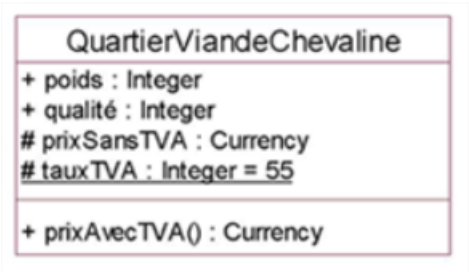
Cette représentation complète intéresse surtout le développeur chargé de la réalisation du logiciel en aval de la modélisation. Il dispose ainsi d'une description de la classe très proche de celle qu'il doit utiliser dans son langage de programmation.

Les attributs et les méthodes de classe

Chaque instance d'une classe contient une valeur spécifique pour chacun de ses attributs. Cette valeur n'est donc pas partagée par l'ensemble des instances. Dans certains cas, il est nécessaire d'utiliser des attributs dont la valeur est commune à l'ensemble des objets d'une classe. Un tel attribut voit sa valeur partagée au même titre que son nom, son type et sa valeur par défaut. Un tel attribut est appelé attribut de classe car il est lié à la classe.

Un attribut de classe est représenté par un nom souligné. Il peut être encapsulé et posséder un type. Il est vivement recommandé de lui donner une valeur par défaut.

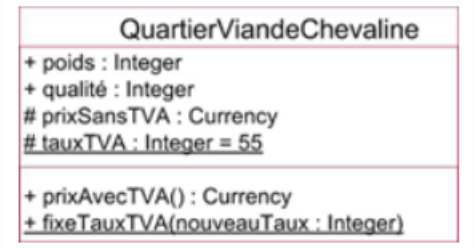
Exemple : Nous étudions un système qui est une boucherie exclusivement chevaline. La figure ci-dessous introduit une nouvelle classe qui décrit un quartier de viande de cheval. Lorsque ce produit est vendu, il est soumis à une TVA dont le montant est le même pour tous les quartiers. Cet attribut est souligné et protégé car il sert à calculer le prix avec TVA incluse. Il est exprimé en pourcentage d'où son type Integer. La valeur par défaut est 55, soit 5,5 %, le taux de TVA des produits alimentaires en France.



Le type utilisé pour l'attribut **prixSansTVA** et le résultat de la méthode **prixAvecTVA** est **Currency**. Il indique que les valeurs sont des montants monétaires.

Au sein d'une classe, il peut également exister une ou plusieurs méthodes de classe. Celles-ci sont liées à la classe. Pour appeler une méthode de classe, il faut envoyer un message à la classe elle-même, et non à l'une de ses instances. Les seuls attributs qu'une telle méthode manipule sont les attributs de classe.

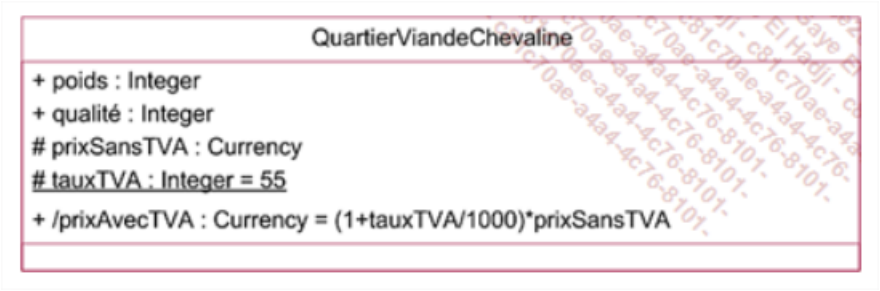
Exemple : La figure suivante ajoute une méthode de classe à la classe **QuartierViandeChevaline** qui sert à fixer le taux de TVA. En effet, celui-ci est fixé par la loi et cette dernière peut être modifiée.



Remarques : Dans de nombreux outils UML, les termes "attribut de classe" et "méthode de classe" ne sont pas utilisés. Ces outils préfèrent les termes de "attribut statique" ou "méthode statique". Ces termes sont employés dans les langages de programmation comme C++ ou Java.

Un attribut ou une méthode de classe ne sont pas hérités. En effet, l'héritage s'applique à la description des instances qui est calculée par l'union de la structure et du comportement de la classe et de ses surclasses. Une sous-classe peut accéder à un attribut ou à une méthode de classe de l'une de ses surclasses, mais n'en hérite pas. S'il y avait héritage, il y aurait autant d'exemplaires de tels attributs ou méthodes que la classe qui les introduit possède de sous-classes.

UML introduit la notion d'attribut calculé dont la valeur est donnée par une fonction basée sur la valeur d'autres attributs. Un tel attribut possède un nom précédé du signe / et il est suivi d'une expression donnant le moyen de calculer sa valeur.



Associations entre objets

Dans le monde réel, de nombreux objets sont liés entre eux. Ce lien correspond à une association qui existe entre les objets.

Exemples :

- Le lien qui existe entre le poulain Espiègle et son père ;
- Le lien qui existe entre le poulain Espiègle et sa mère ;
- Le lien qui existe entre la jument Jorphée et l'élevage de chevaux auquel elle appartient ;
- Le lien qui existe entre l'élevage de chevaux Heyde et son propriétaire.

En UML, ces liens sont décrits par une association, comme un objet est décrit par une classe. Un lien est une occurrence d'une association.

Par conséquent, une association relie des classes. Chaque occurrence de cette association relie entre elles des instances de ces classes.

Une association porte un nom. Comme pour une classe, ce nom est significatif des occurrences de l'association.

Exemples

- L'association père entre la classe Descendant et la classe Étalon ;
- L'association mère entre la classe Descendant et la classe Jument ;
- L'association appartient entre la classe **Cheval** et la classe **ÉlevageChevaux** ;
- L'association propriétaire entre la classe **ÉlevageChevaux** et la classe **Personne**.

Remarque : Les associations que nous avons examinées jusqu'à présent, à titre d'exemple, relient deux classes. De telles associations sont appelées associations binaires. Une association reliant trois classes est appelée association ternaire. Une association reliant n classes est appelée association n-aire. Dans la pratique, la très grande majorité des associations sont binaires et les associations quaternaires et au-delà ne sont quasiment jamais utilisées.

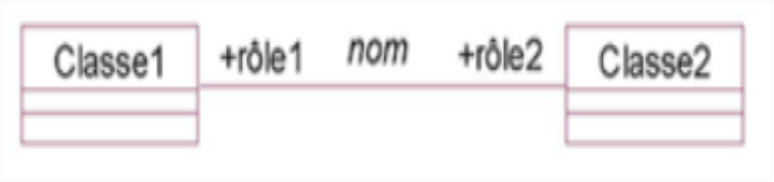
La représentation graphique d'une association binaire consiste en un trait continu entre les deux classes dont elle associe les instances. Ces classes se situent aux extrémités de l'association.

La figure suivante illustre la représentation d'une association binaire. Le nom de l'association est indiqué au-dessus du trait.

Remarque : Il est possible de faire précéder le nom d'une association du signe < ou de le faire suivre par le signe > pour indiquer le sens de lecture du nom vis-à-vis du nom des classes. Si l'association est située sur un axe vertical, il est possible de faire précéder le nom par ^ ou v.

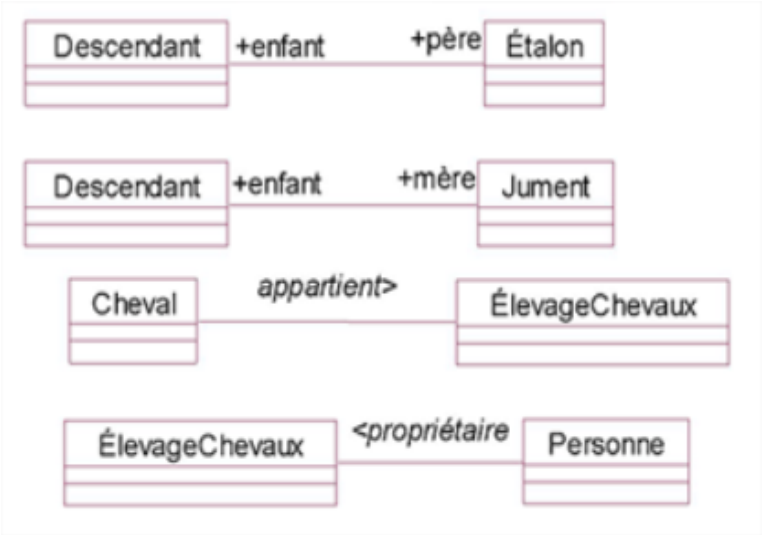
Chaque extrémité d'une association peut également être nommée. Ce nom est significatif du rôle que jouent les instances de la classe correspondante dans l'association. Un rôle a la même nature qu'un attribut dont le type serait la classe située à l'autre extrémité. Par conséquent, il peut être public ou encapsulé de façon privée, protégée ou visible uniquement dans le paquetage. Lorsque les rôles sont précisés, il n'est souvent plus nécessaire d'indiquer le nom de l'association car celui-ci est fréquemment le même que l'un des rôles.

La figure suivante illustre la représentation d'une association binaire en montrant les rôles.

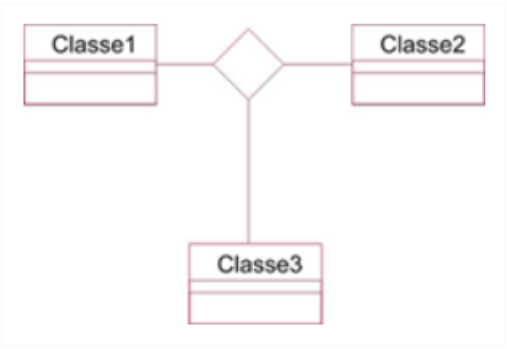


Exemple : La figure suivante illustre la représentation graphique des associations introduites précédemment en exemple.

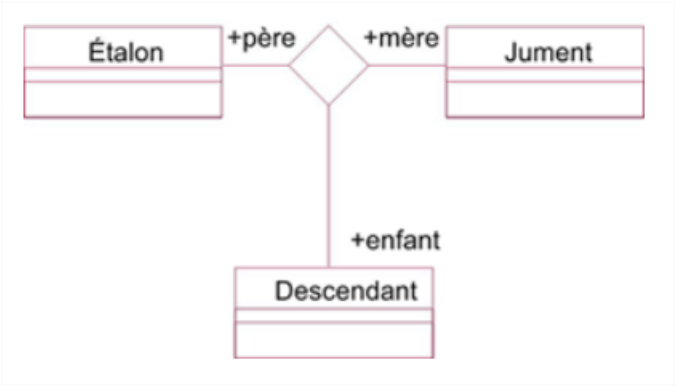
Pour ces associations, soit le nom de l'association, soit ses rôles ont été indiqués.



La représentation graphique d'une association ternaire et au-delà consiste en un losange qui relie les différentes classes. La figure suivante illustre la représentation d'une association ternaire.



Exemple : L'association famille qui relie les classes Étalon, Jument et Descendant est illustrée à la figure ci-dessous. Chacune de ses occurrences constitue un triplet (père, mère, poulain).



La cardinalité située à une extrémité d'une association indique à combien d'instances de la classe située à cette extrémité, une instance de la classe située à l'autre extrémité est liée.

Il est possible de spécifier, à une extrémité d'une association, la cardinalité minimale et la cardinalité maximale pour indiquer un intervalle de valeurs auquel doit toujours appartenir la cardinalité.

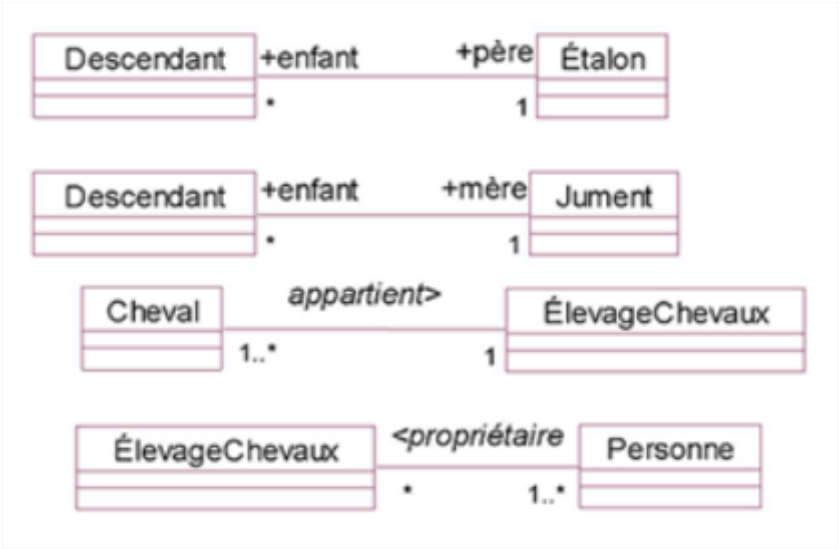
La syntaxe de spécification des cardinalités minimale et maximale est décrite dans le tableau ci-après.

Spécification	Cardinalités
0..1	zéro ou une fois
1	une et une seule fois
*	de zéro à plusieurs fois
1..*	de une à plusieurs fois
M..N	entre M et N fois
N	N fois

En l'absence de spécification explicite des cardinalités minimale et maximale, celles-ci valent 1.

Exemple : À la figure suivante, nous reprenons les associations de cardinalités minimale et maximale de chaque association. Un élevage peut avoir plusieurs copropriétaires et une personne peut être propriétaire de plusieurs élevages.

À titre d'exemple, la première association se lit ainsi : un descendant possède un seul père, un étalon peut avoir de zéro à plusieurs enfants.

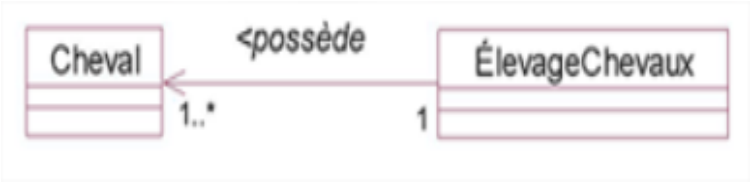


Les associations ont par défaut une navigation bidirectionnelle, c'est-à-dire qu'il est possible de déterminer les liens de l'association depuis une instance de chaque classe d'origine. Une navigation bidirectionnelle est plus complexe à réaliser par les développeurs ; il convient de l'éviter dans la mesure du possible.

Spécifier le seul sens de navigation utile lors des phases de la modélisation proches du passage au développement se fait en dessinant l'association sous forme d'une flèche.

Exemple : Dans le cadre particulier d'un élevage de chevaux, il est utile de connaître les chevaux que cet élevage possède, mais le contraire, à savoir connaître les élevages qui possèdent un cheval, n'est pas utile.

La figure suivante illustre l'association résultant avec le sens de navigation adapté.



Une association réflexive est une association où la même classe peut se trouve à chaque extrémité d'une association. Il s'agit alors d'une association réflexive, reliant entre elles les instances d'une même classe.

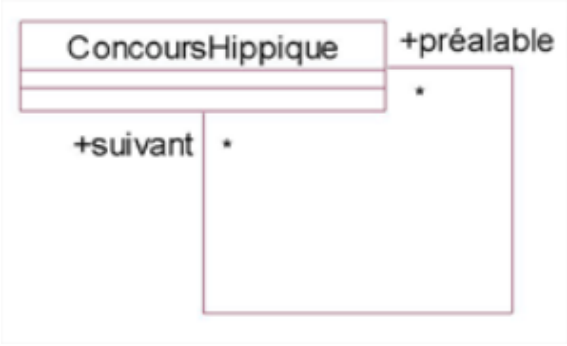
Dans ce cas, il est préférable de nommer le rôle joué par la classe à chaque extrémité de l'association.

Comme nous le verrons dans les exemples, une association réflexive sert principalement à décrire au sein de l'ensemble des instances d'une classe :

- Des groupes d'instances ;
- Ou une hiérarchie au sein des instances.

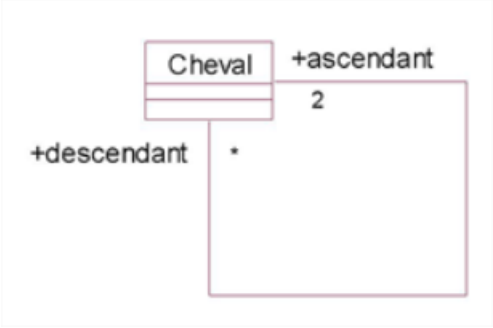
Remarque : Pour les experts, dans le premier cas, il s'agit d'une association représentant une relation d'équivalence et, dans le second, d'une association représentant une relation d'ordre.

Exemple : Pour pouvoir passer les épreuves de sélection d'un concours hippique international, un cheval doit avoir gagné d'autres concours préalables. Il est donc possible de créer une association entre un concours et ses concours préalables. La figure suivante illustre cette association. Cette association crée une hiérarchie au sein des concours.

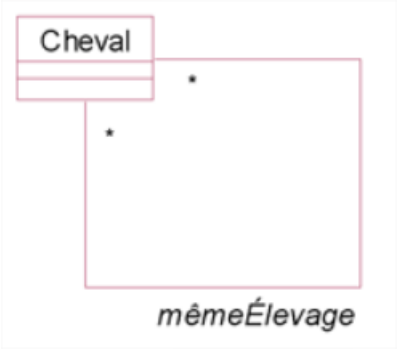


Exemple : La figure suivante illustre l'association "ascendant/descendant direct" entre les chevaux. Cette association crée une hiérarchie au sein des chevaux.

Pour les ascendants, la cardinalité est 2 car tout cheval a exactement une mère et un père.



Exemple : La figure suivante illustre l'association entre les chevaux qui se trouvent dans le même élevage. Cette association crée des groupes au sein de l'ensemble des instances de la classe Cheval, chaque groupe correspondant à un élevage.

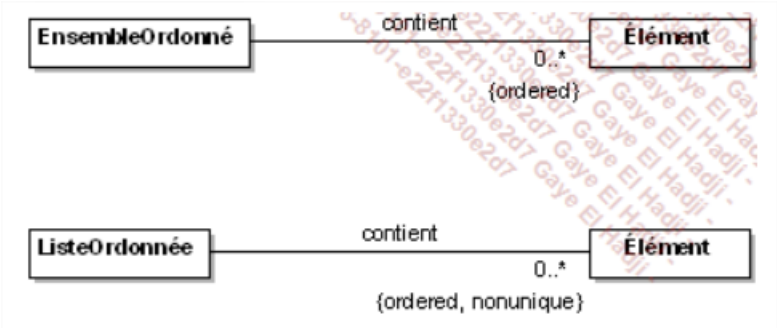


Les extrémités des associations peuvent posséder des propriétés à l'image des attributs. Les principales propriétés des extrémités des associations sont les suivantes :

{ordered} : lorsqu'une extrémité possède une cardinalité supérieure à 1, les occurrences doivent être ordonnées.

{nonunique} : cette propriété permet d'indiquer qu'une instance située à l'extrémité où se trouve la propriété peut être liée plusieurs fois à une instance située à l'autre extrémité. La cardinalité doit être supérieure à 1. Cette propriété n'est pas utilisée par défaut.

Exemple : La figure suivante illustre l'utilisation des deux propriétés. La partie supérieure montre la description d'un ensemble ordonné. Dans un ensemble, un même élément ne peut pas apparaître plus d'une fois. À l'opposé, dans une liste, un même élément peut apparaître plusieurs fois. Dans la partie inférieure de la figure suivante, une liste ordonnée est décrite. La propriété {nonunique} est donc utilisée.

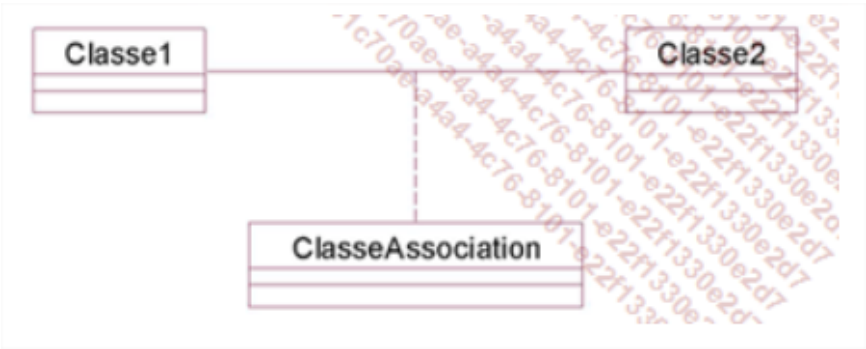


Les liens entre les instances des classes peuvent porter des informations. Celles-ci sont spécifiques à chaque lien.

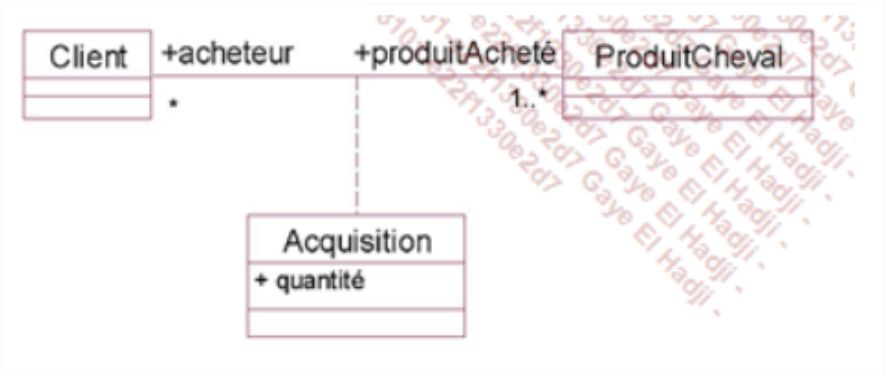
Dans ce cas, l'association qui décrit de tels liens reçoit le statut de classe, dont les instances sont des occurrences de l'association.

Comme toute autre classe, une telle classe peut être dotée d'attributs, d'opérations, et être reliée au travers d'associations à d'autres classes.

La figure suivante représente graphiquement une classe-association. Celle-ci est reliée à l'association par un trait pointillé.



Exemple : Quand un client achète un produit pour cheval (produits d'entretien, etc.), il convient de spécifier la quantité de produits acquis par une classe association, ici la classe Acquisition.



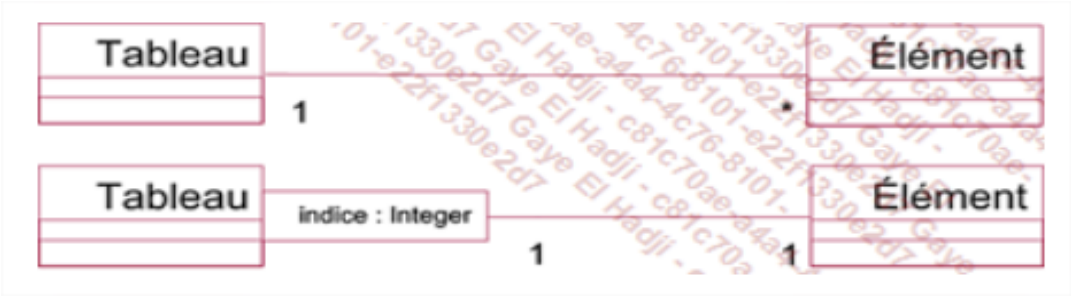
En cas de cardinalité maximale non finie à une extrémité d'une association, si les instances situées à cette extrémité sont qualifiables, il est possible d'utiliser cette qualification pour passer de la cardinalité maximale non finie à une cardinalité maximale finie.

La qualification d'une instance est une valeur ou un ensemble de valeurs qui permettent de retrouver cette instance. Une qualification est souvent un index, par exemple pour retrouver un élément dans un tableau, ou une clé, par exemple pour retrouver une ligne dans une base de données relationnelle.

Cette qualification est alors insérée à l'extrémité opposée à celle où se trouve la cardinalité maximale. Cette qualification se présente sous la forme d'un ou de plusieurs attributs (voir figure ci-dessous) qui qualifient les instances de la classe 2 au niveau de la classe 1.

Exemple : Un tableau est constitué d'éléments. La figure suivante décrit deux possibilités de modélisation en UML. La première ne fait pas appel à la qualification tandis que, dans la seconde,

le qualificateur indice permet de retrouver un seul élément du tableau ; par conséquent, la cardinalité maximale passe à 1.



Exemple : Lors d'une course de chevaux, chaque cheval est numéroté. La figure suivante illustre l'ensemble des participants d'une course en les qualifiant par leur numéro.

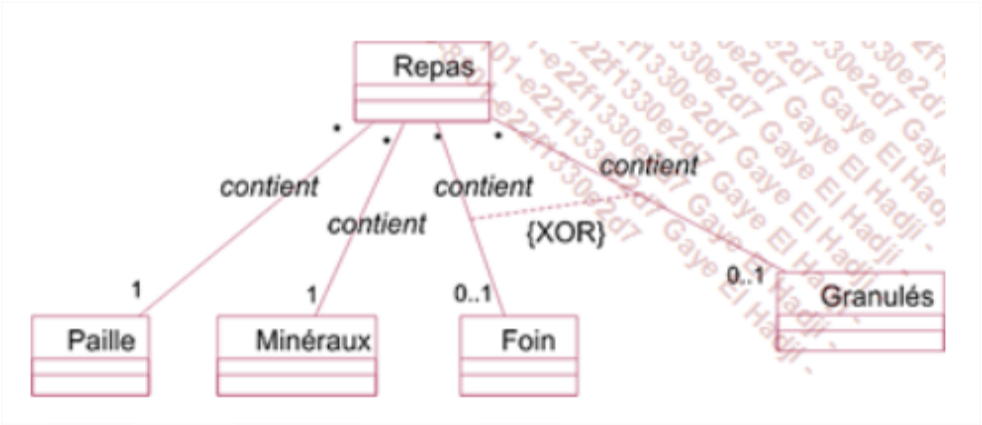
UML offre l'expression des contraintes, grâce à certaines constructions de la modélisation objet que nous avons déjà étudiées : les cardinalités, le type d'un attribut, etc.

UML introduit une contrainte appelée {XOR} (ou exclusif) entre deux associations qui possèdent au moins à l'une de leurs extrémités une classe commune. La contrainte exprime que chaque instance de la classe commune ne peut pas participer aux deux associations.

Exemple : Un repas pour un cheval contient les éléments suivants :

- Paille ;
- Minéraux ;
- Foin ou granulés.

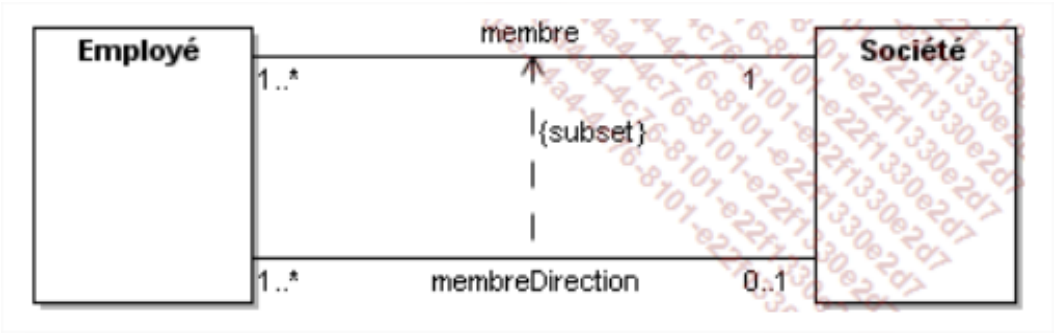
La figure 6.29 illustre le repas et ses différents constituants. Qu'il contienne soit du foin, soit des granulés est une contrainte exprimée par l'opérateur XOR. Celui-ci exprime la contrainte du ou exclusif entre les deux associations.



Une autre contrainte peut être spécifiée entre deux associations qui possèdent les mêmes classes à chaque extrémité. Il s'agit de la contrainte {subset} (sous-ensemble) qui permet d'indiquer que l'ensemble des occurrences de l'une des associations est inclus dans l'ensemble des occurrences de l'autre association. La représentation graphique est comparable à celle de la contrainte {XOR} avec une ligne pointillée orientée allant de l'association de type sous-ensemble vers l'autre association.

Exemple : L'association **membreDirection** entre une société et ses employés est une association de type sous-ensemble de l'association membre entre les employés et ladite société.

La figure suivante illustre la contrainte **{subset}** entre ces deux associations.



Cependant, les contraintes de ce type peuvent se révéler insuffisantes. Pour d'autres contraintes, UML propose de les exprimer en langage naturel ou en OCL.

OCL est un langage de contraintes objet sous forme de conditions logiques. Il fait partie de l'ensemble de la notation UML.

Qu'elles soient écrites en langage naturel ou en OCL, les contraintes sont représentées dans des notes incluses à l'intérieur du diagramme de classes.

Les contraintes écrites en OCL s'expriment sur la valeur des attributs et des rôles (extrémités des associations). Une contrainte doit avoir une valeur logique (vrai ou faux).

Pour construire une contrainte, on utilise tous les opérateurs applicables sur la valeur des attributs en fonction de leur type (comparaison d'entiers, addition d'entiers, comparaison de chaînes, etc.). Il est possible d'employer les méthodes des objets au sein des conditions. Pour les valeurs

ensemblistes (ensembles d'objets), OCL propose un jeu d'opérateurs : union, intersection, différence. Pour les collections d'objets, OCL propose notamment les opérateurs suivants : collect, includes, includesAll, asSet, exists, forAll.

Pour désigner un attribut ou une méthode dans une expression OCL, il faut élaborer une expression de chemin qui commence par self. Ensuite, on désigne directement l'attribut ou la méthode par son nom ou l'on traverse une association en utilisant le nom du rôle. Il convient alors soit de désigner un attribut ou une méthode de la classe située à l'autre extrémité de l'association, soit de désigner à nouveau un rôle pour traverser une autre association jusqu'au choix d'un attribut ou d'une méthode.

Remarque : Traverser une association consiste à partir d'une extrémité de celle-ci pour retrouver les instances présentes à l'autre extrémité.

La syntaxe d'une expression de chemin est la suivante :

self.attribut

ou

self.méthode

ou

self.rôle.rôle. ... .rôle.attribut

ou

self.rôle.rôle. ... .rôle.méthode

Si une contrainte OCL n'est pas directement incluse dans le diagramme de classes, alors il faut préciser son contexte selon la syntaxe suivante :

Context Classe **inv** Contrainte :

Remarque : Nous ne présentons dans cet ouvrage qu'une introduction au langage OCL. Les lecteurs désireux d'obtenir plus d'informations pourront se référer à l'ouvrage The Object Constraint Language : Getting Your Models ready for MDA, Second Edition de Jos Warmer et Anneke Kleppe, Addison-Wesley, 2003.

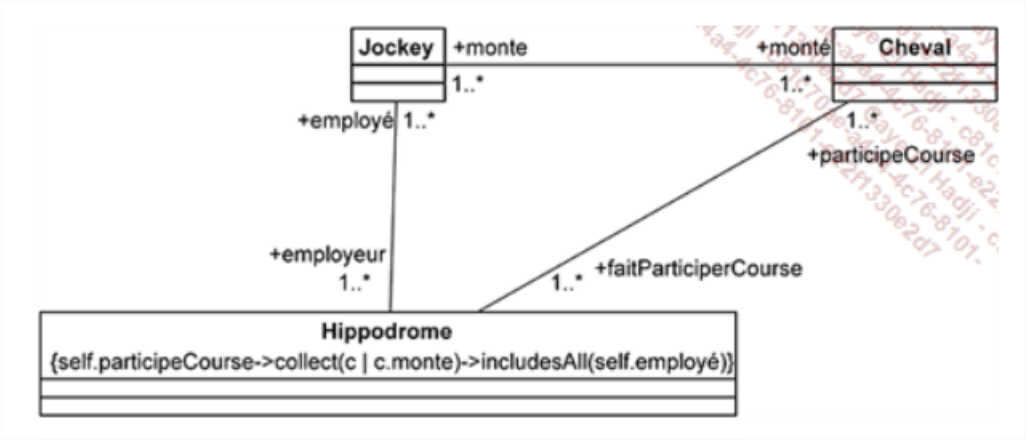
Exemple : Un hippodrome fait courir des chevaux. Les jockeys dont il est l'employeur doivent participer aux courses qu'il organise. Ceci peut être reformulé ainsi : l'ensemble des jockeys employés par l'hippodrome doit être inclus dans l'ensemble de tous les jockeys qui montent les chevaux participant aux courses de l'hippodrome.

En OCL, cette contrainte s'écrit ainsi :

Context Hippodrome **inv** jockeysEmployés:
self.participeCourse->collect(c | c.monte)->includesAll

(self.employé)

Une telle contrainte s'appuie sur l'utilisation des expressions de chemin en OCL. La figure 6.31 illustre le diagramme de classes où la contrainte écrite en OCL est directement incluse.



Les objets composés

Un objet peut être composé d'autres objets. Dans ce cas, il s'agit d'une association entre objets appelée association de composition. Elle associe un objet complexe aux objets qui le constituent, à savoir ses composants.

Il existe deux formes de compositions : forte ou faible.

La composition forte ou composition

La composition forte est la forme de composition telle que les composants sont une partie de l'objet composé. Chaque composant ne peut ainsi être partagé entre plusieurs objets composés. La cardinalité maximale, au niveau de l'objet composé, est donc obligatoirement 1.

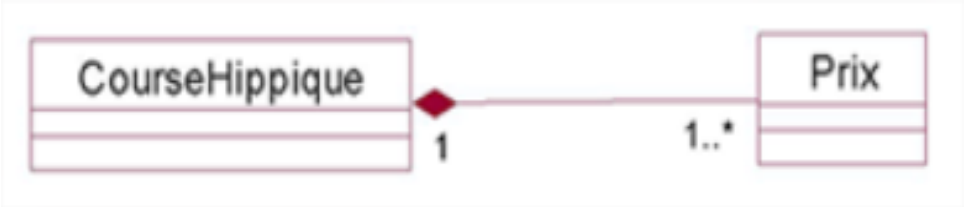
La suppression de l'objet composé entraîne la suppression de ses composants.

La figure ci-dessous montre la représentation graphique de l'association de composition forte. Au niveau de l'objet composé, la cardinalité minimale indiquée est 0 mais elle pourrait aussi être 1.

L'association de composition forte sera appelée plus simplement composition.

Exemple : Un cheval est composé d'un cerveau. Le cerveau n'est pas partagé. La mort du cheval entraîne la mort de son cerveau. Il s'agit donc d'une association de composition. Celle-ci est illustrée à la figure suivante.

Exemple : Une course hippique est constituée de prix. Ces prix ne sont pas partagés avec d'autres courses (un prix est spécifique à une course). Si la course n'est pas organisée, les prix ne sont pas attribués et ils disparaissent. Il s'agit d'une relation de composition, illustrée à la figure suivante.



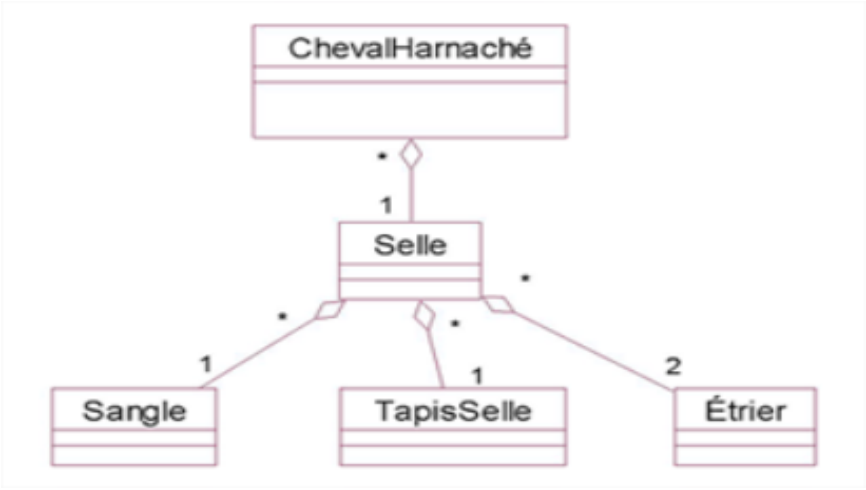
La composition faible ou agrégation

La composition faible, appelée plus couramment agrégation, impose beaucoup moins de contraintes aux composants que la composition forte. Dans le cas de l'agrégation, les composants peuvent être partagés par plusieurs composés (de la même association d'agrégation ou de plusieurs associations distinctes d'agrégation) et la destruction du composé ne conduit pas à la destruction des composants.

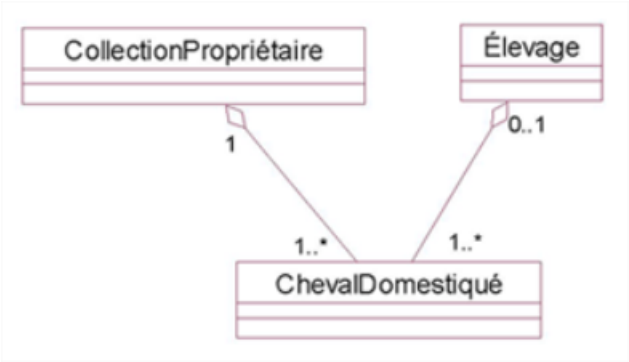
L'agrégation se rencontre plus fréquemment que la composition. Lors d'une première phase de modélisation, il est possible d'utiliser seulement l'agrégation puis, plus tard, de déterminer quelles associations d'agrégation sont des associations de composition.

Remarque : Déterminer, sur un modèle, qu'une association d'agrégation est une association de composition, revient à ajouter des contraintes, au même titre que donner un type ou préciser des cardinalités. Nous avons étudié les contraintes en UML, en langage naturel ou en OCL. Ajouter des contraintes, c'est ajouter du sens, de la sémantique à un modèle, c'est l'enrichir. Il est donc normal que ce processus d'enrichissement requière des phases successives.

Exemple : Un cheval harnaché est composé d'une selle. Une selle est elle-même composée d'une sangle, d'étriers et d'un tapis de selle. Cette composition relève de l'agrégation (voir figure ci-dessous). En effet, la perte du cheval n'entraîne pas la perte de ces objets et la perte de la selle n'entraîne pas la perte de ses composants.



Exemple : Un propriétaire équestre possède une collection de chevaux. Un cheval domestiqué appartient à une seule collection et peut simultanément être confié ou non à un élevage. Il peut être alors composant des deux agrégations. Ces deux associations sont illustrées à la figure suivante.



Exemple : De façon plus précise, un cheval peut appartenir à plusieurs propriétaires. Dans ce cas, la cardinalité au niveau de la collection du propriétaire n'est plus 1 mais 1..* pour exprimer la multiplicité. Le cheval peut être alors partagé plusieurs fois dans une même agrégation (voir figure ci-dessous).

Les différences entre composition et agrégation

Le tableau suivant résume l'ensemble des différences entre agrégation et composition.

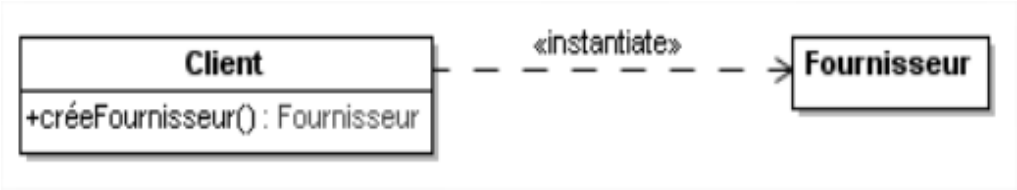
	Agrégation	Composition
Représentation	losange transparent	losange noir
Partage des composants par plusieurs associations	oui	non
Destruction des composants lors de la destruction du composé	non	oui
Cardinalité au niveau du composé	quelconque	0..1 ou 1

La relation de dépendance

La relation de dépendance indique qu'une classe cliente a besoin d'une autre classe fournisseur pour sa propre spécification ou pour sa réalisation.

Graphiquement, la relation de dépendance est représentée par une flèche pointillée allant de la classe Client à la classe Fournisseur.

Exemple : La classe Client possède une méthode dont le type de l'un des paramètres ou le type de retour est la classe Fournisseur. La figure suivante montre un tel exemple.



Une relation de dépendance peut être munie d'un stéréotype pour définir sa sémantique. Les stéréotypes les plus importants sont les suivants :

«**call**» : l'implantation de la classe Client invoque une méthode de la classe Fournisseur.

«**create**» : l'implantation de la classe Client crée une instance de la classe Fournisseur pour un usage interne.

«**derive**» : la spécification et l'implantation de la classe Client sont uniquement obtenues à partir de la spécification et de l'implantation de la classe Fournisseur. Le client est donc une redondance qui a été construite, par exemple, pour des raisons d'optimisation ou de facilité d'utilisation.

«**instantiate**» : le client est une fabrique d'objets du fournisseur. Une fabrique est une classe dont le seul but est de créer des instances d'une ou de plusieurs autres classes. La figure 6.38 illustre une fabrique. En effet, la classe Client possède une méthode publique créeFournisseur qui crée une nouvelle instance de la classe Fournisseur.

«**permit**» : la classe Fournisseur autorise la classe Client à accéder à plusieurs ou à la totalité de ses attributs privés et/ou de ses méthodes privées.

«**refine**» : la classe Client spécialise la classe Fournisseur. L'utilisation de ce stéréotype à la place de la relation de spécialisation est souvent mise en œuvre lors des premières étapes de la conception des diagrammes UML d'un système, avant que cette relation soit remplacée par la relation de spécialisation.

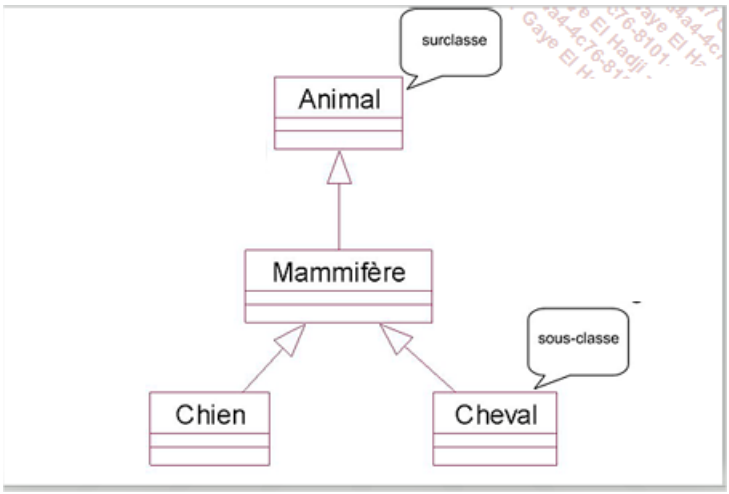
«**use**» : il s'agit du stéréotype plus général qui spécifie que la classe Client a besoin, sans fournir plus de précisions, de la classe Fournisseur.

La relation de généralisation/spécialisation entre les classes

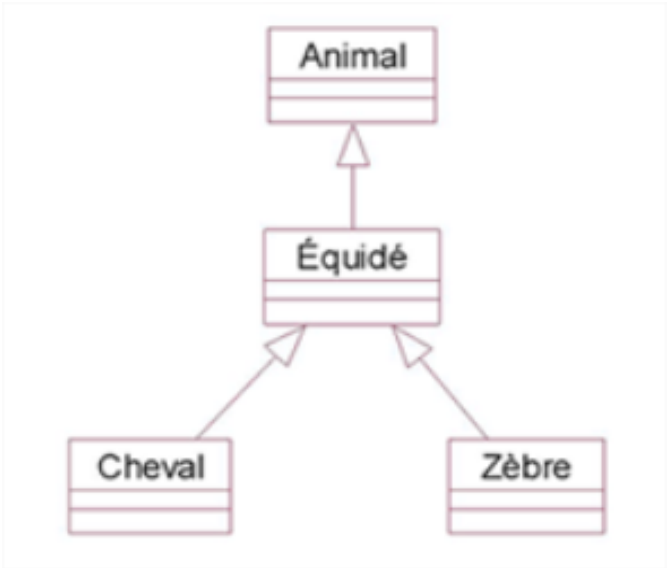
Les classes plus spécifiques et les classes plus générales

Une classe est plus spécifique qu'une autre si toutes ses instances sont également instances de cette autre classe. La classe plus spécifique est dite sous-classe de l'autre classe. Cette dernière, plus générale, est dite surclasse.

La figure suivante illustre ces deux classes ainsi que la relation de généralisation/spécialisation qui les relie.



Exemple : Le cheval est une spécialisation de l'équidé, elle-même spécialisation de l'animal. Le zèbre est une autre spécialisation de l'équidé. Le résultat est la petite hiérarchie illustrée à la figure suivante.



L'héritage

Les instances d'une classe sont aussi instances de sa ou de ses surclasses. En conséquence, elles profitent des attributs et des méthodes définis dans la ou les surclasses, en plus des attributs et des méthodes introduits au niveau de leur classe.

Cette faculté s'appelle l'héritage, c'est-à-dire qu'une classe hérite des attributs et méthodes de ses surclasses pour en faire bénéficier ses instances.

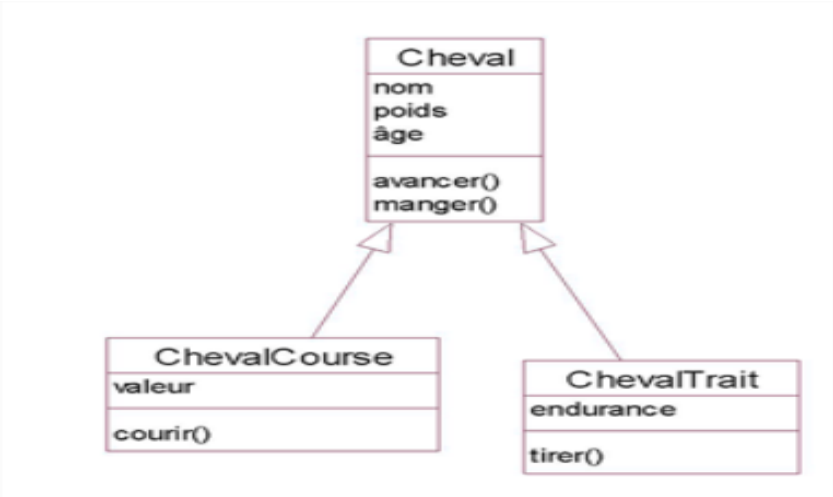
UML offre la possibilité de représenter les attributs et les méthodes hérités dans les sous-classes en les faisant précéder du symbole ^.

Remarques : Rappelons que les attributs et méthodes privés d'une surclasse sont hérités dans ses sous-classes mais n'y sont pas visibles.

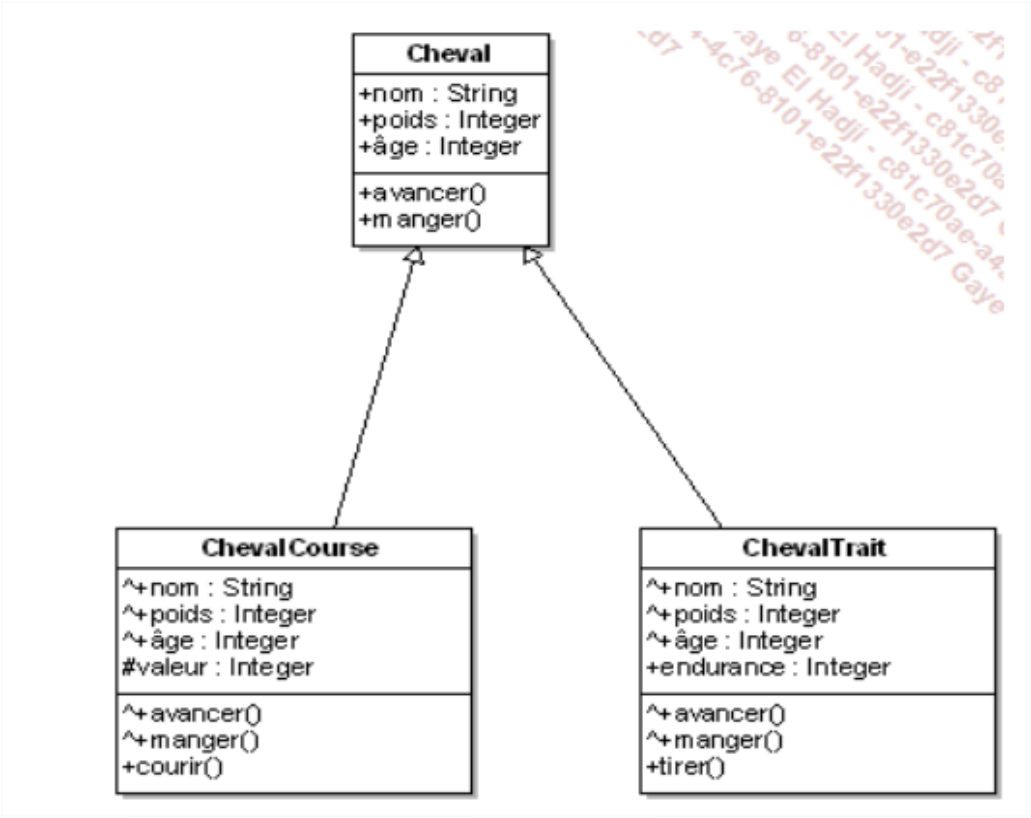
Lors d'un héritage, une méthode ou un attribut peuvent être redéfinis dans la sous-classe. Cette redéfinition s'applique principalement dans le cas de l'héritage d'une classe abstraite.

Exemple : Les attributs et les méthodes de la classe Cheval sont hérités dans ses deux sous-classes, comme l'illustre la figure ci-dessous.

Cet héritage signifie qu'un cheval de course, comme un cheval de trait, possède un nom, un poids, un âge et qu'il peut avancer et manger.



La figure suivante fait apparaître la même hiérarchie en indiquant les attributs et les méthodes hérités qui sont alors précédés du symbole ^. Le type et l'encapsulation des attributs et des méthodes ont également été ajoutés.


Cheval
+nom : String
+poids : Integer
+âge : Integer
+avancer()
+manger()
ChevalCourse
^+nom : String
^+poids : Integer
^+âge : Integer
#valeur : Integer
^+avancer()
^+manger()
+courir()
ChevalTrait
^+nom : String
^+poids : Integer
^+âge : Integer
+endurance : Integer
^+avancer()
^+manger()
+tirer()

Les classes concrètes et abstraites

La figure ci-dessous montre l'existence de deux types de classes dans la hiérarchie, à savoir les classes concrètes Cheval et Loup, qui apparaissent tout en bas de la hiérarchie, ainsi que la classe abstraite Animal.

Une classe concrète possède des instances. Elle constitue un modèle complet d'objets (tous les attributs et méthodes sont complètement décrits).

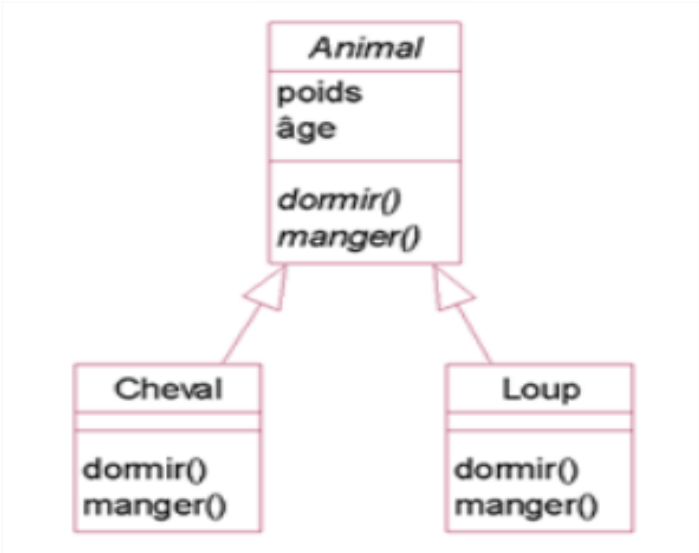
À l'opposé, une classe abstraite ne peut pas posséder d'instance directe car elle ne fournit pas une description complète. Elle a pour vocation de posséder des sous-classes concrètes et sert à factoriser des attributs et méthodes communs à ses sous-classes.

Souvent, la factorisation de méthodes communes aux sous-classes se traduit par la seule factorisation de la signature. Une méthode introduite dans une classe avec sa seule signature et sans code est appelée une méthode abstraite.

En UML, une classe ou une méthode abstraite sont représentées par le stéréotype «abstract». Graphiquement, celui-ci est représenté soit explicitement, soit implicitement avec une mise en italique du nom de la classe ou de la méthode.

Exemple : Un animal peut dormir ou manger, mais de façon distincte selon la nature de l'animal. Ces méthodes possèdent pour unique description, au niveau de la classe Animal, leur signature. Il s'agit donc de méthodes abstraites.

La figure ci-dessous illustre ces méthodes abstraites au sein de la classe abstraite Animal. Elle montre aussi leur redéfinition pour les rendre concrètes (c'est-à-dire leur attribuer du code) dans les classes Cheval et Loup. En effet, un cheval dort debout alors qu'un loup dort couché. Ils ne mangent pas de la même façon : un loup est carnivore alors qu'un cheval est un herbivore.



Remarque : La signature est l'ensemble constitué du nom de la méthode, des paramètres avec leur nom et leur type, ainsi que du type du résultat, à l'exclusion du code de la méthode.

Toute classe possédant au moins une méthode abstraite est une classe abstraite. En effet, la seule présence d'une méthode incomplète (le code est absent) implique que la classe n'est pas une description complète d'objets.

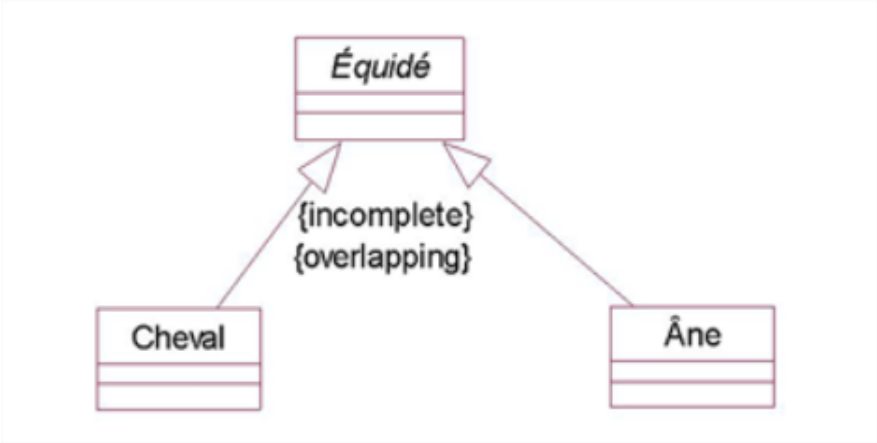
Une classe peut être abstraite même si elle n'introduit aucune méthode abstraite.

Contraintes sur la relation d'héritage

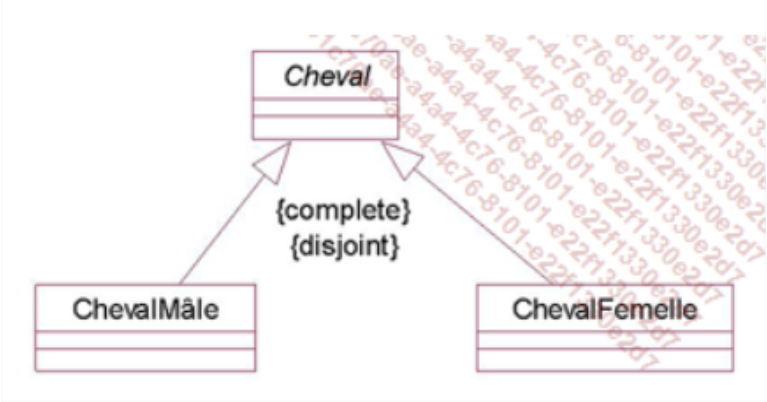
UML offre quatre contraintes sur la relation d'héritage entre une surclasse et ses sous-classes :

- La contrainte {incomplete} signifie que l'ensemble des sous-classes est incomplet et qu'il ne couvre pas la surclasse ou encore que l'ensemble des instances des sous-classes est un sous-ensemble de l'ensemble des instances de la surclasse.
- La contrainte {complete} signifie au contraire que l'ensemble des sous-classes est complet et qu'il couvre la surclasse.
- La contrainte {disjoint} signifie que les sous-classes n'ont aucune instance en commun.
- La contrainte {overlapping} signifie que les sous-classes peuvent avoir une ou plusieurs instances en commun.

Exemple : La figure suivante illustre une relation d'héritage entre la surclasse Équidé et deux sous-classes : Cheval et Âne. Ces deux sous-classes ne couvrent pas la classe des équidés (d'autres sous-classes existent comme les zèbres). Par ailleurs, il existe les mulets, qui sont issus d'un croisement. Ils sont à la fois des chevaux et des ânes. D'où l'utilisation des contraintes {incomplete} et {overlapping}.



Exemple : La figure ci-dessous illustre une autre relation d'héritage entre la surclasse Cheval et deux sous-classes : ChevalMâle et ChevalFemelle. Ces deux sous-classes couvrent la classe des chevaux et il n'existe aucun cheval qui soit à la fois mâle et femelle. D'où l'utilisation des contraintes {complete} et {disjoint}.



Héritage multiple

On parle d'héritage multiple en UML quand une sous-classe hérite de plusieurs surclasses. L'héritage multiple pose un seul problème : il engendre un conflit lorsqu'une même méthode, c'est-à-dire dotée de la même signature, est héritée plusieurs fois dans la sous-classe. En effet, lors de la réception d'un message appelant cette méthode, il faut définir un critère pour en choisir une parmi toutes celles qui sont héritées.

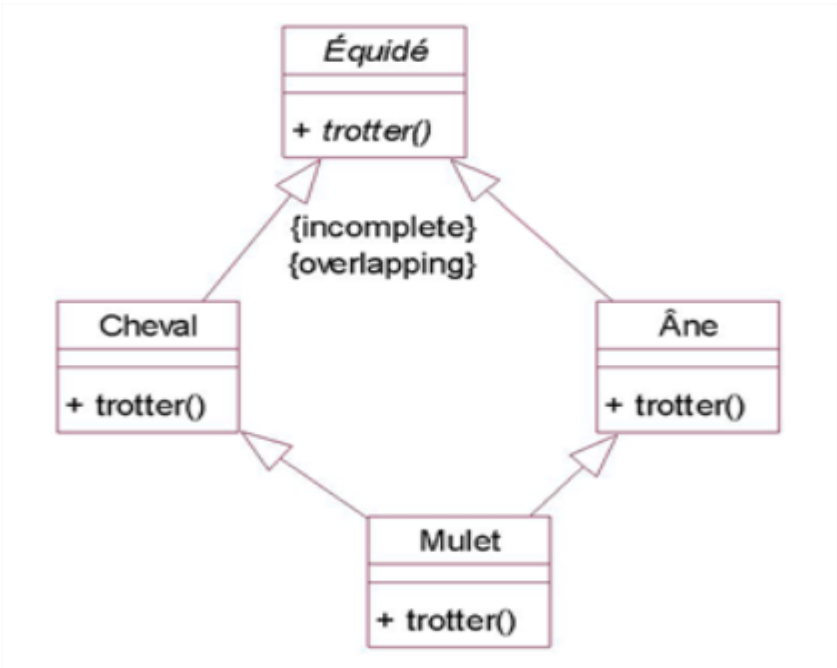
Une solution, dans ce cas, consiste à redéfinir la méthode dans la sous-classe afin de supprimer le conflit.

Le même problème peut se poser pour un attribut de même nom introduit dans plusieurs surclasses. Une solution consiste à utiliser la propriété {redefines nomAttribut} pour le redéfinir dans la sous-classe.

Remarque : Si l'utilisation de l'héritage multiple est bien sûr possible en modélisation, il est souvent nécessaire de transformer les diagrammes de classes pour le supprimer lors du passage au développement. En effet, rares sont les langages de programmation supportant cette forme d'héritage. Pour cela, il existe différentes techniques parmi lesquelles la transformation de chaque héritage multiple en une agrégation.

Exemple : La figure ci-dessous reprend les deux sous-classes Cheval et Âne et introduit une sous-classe commune, à savoir Mulet.

La figure ci-dessous illustre également la méthode **trotter**, abstraite dans la classe Equidé, rendue concrète dans ses sous-classes immédiates, et redéfinie dans la sous-classe fruit de leur héritage multiple. En effet, lorsque son cavalier lui demande de trotter, un mulet répond à la fois par des réactions de cheval et des réactions d'âne. Il peut être dangereux comme un cheval et agaçant comme un âne.

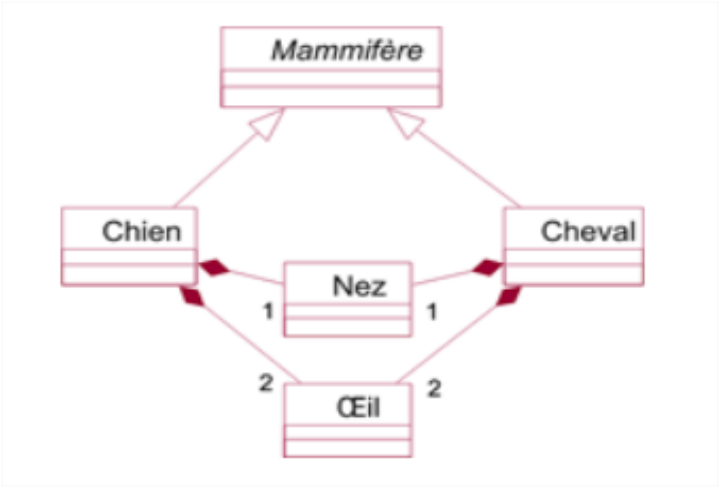


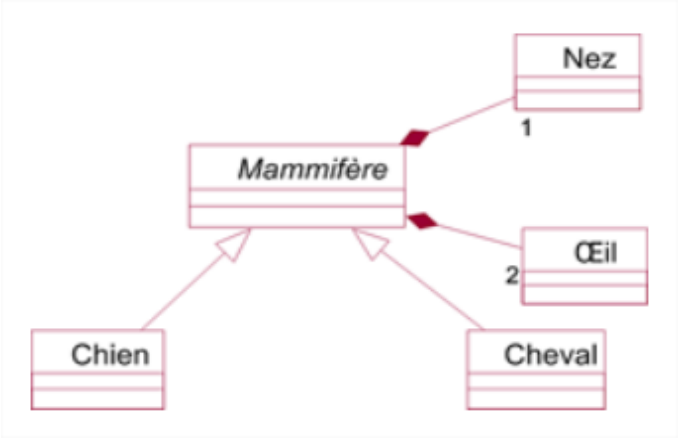
Factorisation des relations entre objets

Une classe abstraite sert à factoriser les attributs et méthodes de plusieurs sous-classes. Il est également possible, parfois, de factoriser l'extrémité d'une association dans une surclasse pour rendre le diagramme plus simple.

Exemple : Un cheval, comme un loup, possède deux yeux et un nez (voir première figure ci-dessous).

La deuxième figure ci-dessous montre qu'une fois créée, la classe abstraite Mammifère surclasse des classes Cheval et Loup, les deux associations de composition sont factorisées.



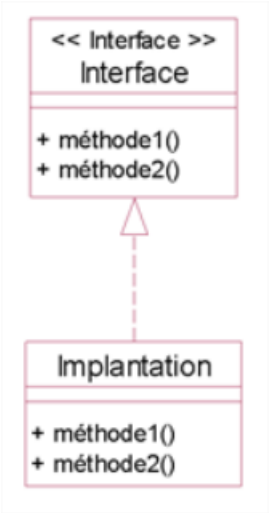


Les interfaces

Une interface est une classe totalement abstraite, c'est-à-dire sans attribut et dont toutes les méthodes sont abstraites et publiques. Une telle classe ne contient aucun élément d'implantation des méthodes. Graphiquement, elle est représentée comme une classe avec le stéréotype «interface».

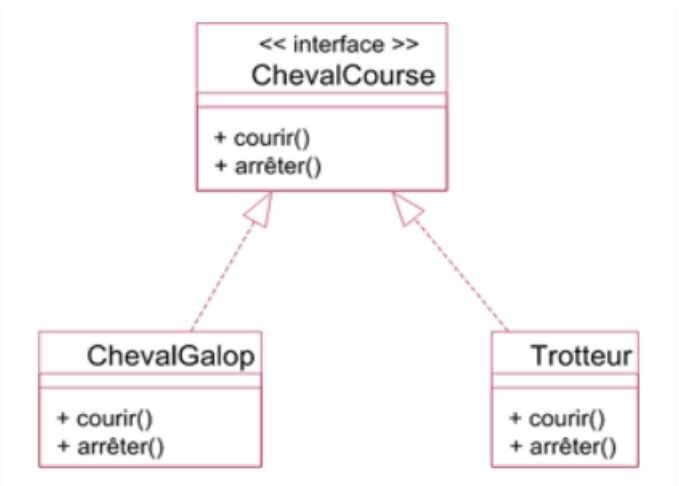
L'implantation des méthodes est réalisée par une ou plusieurs classes concrètes, sous-classes de l'interface. Dans ce cas, la relation d'héritage qui existe entre l'interface et une sous-classe d'implantation est appelée relation de réalisation. Graphiquement, elle est représentée par un trait pointillé, au lieu d'un trait plein pour une relation d'héritage entre deux classes.

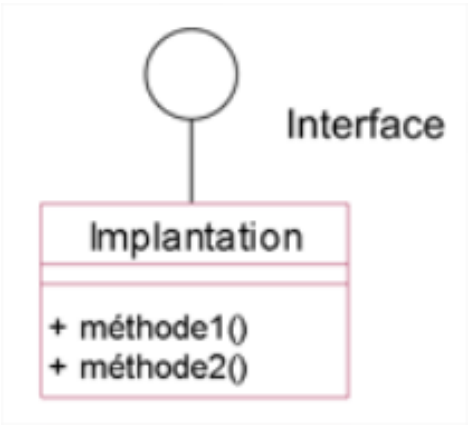
La figure ci-dessous illustre cette représentation.



Exemple : Un cheval de course peut être considéré comme une interface. Celle-ci est composée de plusieurs méthodes : courir, arrêter, etc.

L'implantation peut ensuite différer. Une course de galop ou de trot se fait seulement avec des chevaux entraînés pour l'une ou l'autre de ces courses. Pour des chevaux de galop, courir signifie galoper. Pour des chevaux de trot (un trotteur), courir signifie trotter. Tous deux répondent à l'interface ChevalCourse mais avec une implantation différente due à leur entraînement.



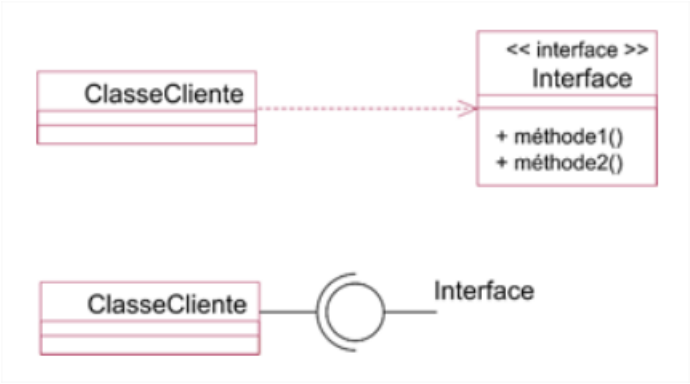


Remarque : Une même classe peut réaliser plusieurs interfaces. Il s'agit d'un cas particulier de l'héritage multiple. En effet, aucun conflit n'est possible car seules les signatures des méthodes sont héritées dans la classe de réalisation. Si plusieurs interfaces contiennent la même signature, cette signature est implantée par une seule méthode dans la classe commune de réalisation.

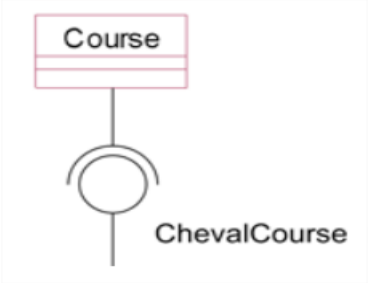
Une classe peut également dépendre d'une interface pour réaliser ses opérations. Cette dernière est alors employée comme type au sein de la classe (attribut, paramètre de l'une des méthodes ou variable locale de l'une des méthodes).

Une classe qui dépend d'une interface en est sa cliente.

La figure ci-dessous illustre la relation de dépendance entre une classe et une interface.



Exemple : Une course a besoin de chevaux de course pour être organisée. La classe Course dépend donc de l'interface ChevalCourse (voir figure ci-dessous).



Représentation des objets ou instances

Le diagramme de classes est une représentation statique du système. Il peut également montrer les objets, c'est-à-dire, à un moment donné, les instances créées et leurs liens lorsque le système est actif.

Chaque instance est représentée dans un rectangle qui contient son nom en style souligné et, éventuellement, la valeur d'un ou de plusieurs attributs.

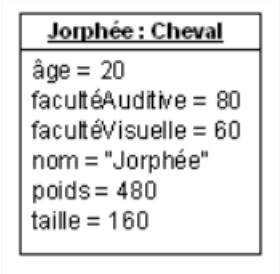
Le nom d'une instance prend la forme :

nomInstance : nomClasse

Le nom de l'instance est optionnel.

La valeur d'un attribut est de la forme :

nomAttribut = valeurAttribut

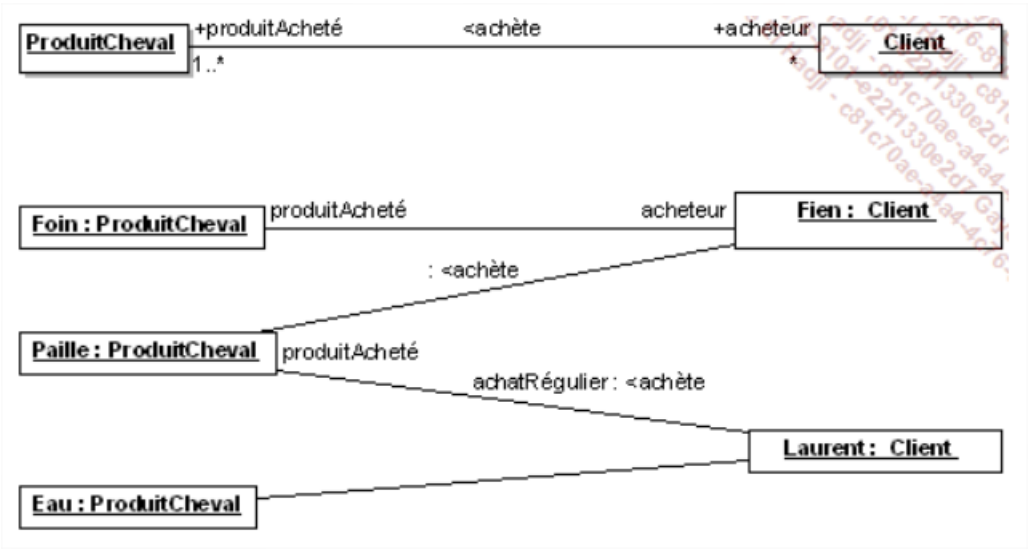


La relation d'instanciation décrit le lien qui existe entre une instance et sa classe. Cette relation est décrite dans le nom de l'instance qui est suffixé par le nom de la classe. Il est également possible de préciser cette relation entre l'instance et sa classe en utilisant une relation de dépendance munie du stéréotype «instanceOf». Cette dernière représentation est toutefois moins courante.

Exemple : La figure ci-dessous montre l'instance Jorphée et sa classe Cheval. La relation d'instanciation est précisée de deux façons. Tout d'abord, le nom Jorphée est suffixé par le nom Cheval. Ensuite, la figure inclut une relation de dépendance entre Jorphée et la classe Cheval. Cette relation est dotée du stéréotype «instanceOf».

Les liens entre instances sont représentés par de simples traits continus. Ces liens sont les occurrences de relations interobjets. De ce fait, à l'image du nom des objets, le nom des liens est représenté graphiquement en style souligné et il est suffixé par le nom de l'association. Il est également possible de présenter le nom des rôles sur la représentation graphique des liens. Ces noms de rôle ne sont pas soulignés car ils ne correspondent ni à des instances ni à des occurrences.

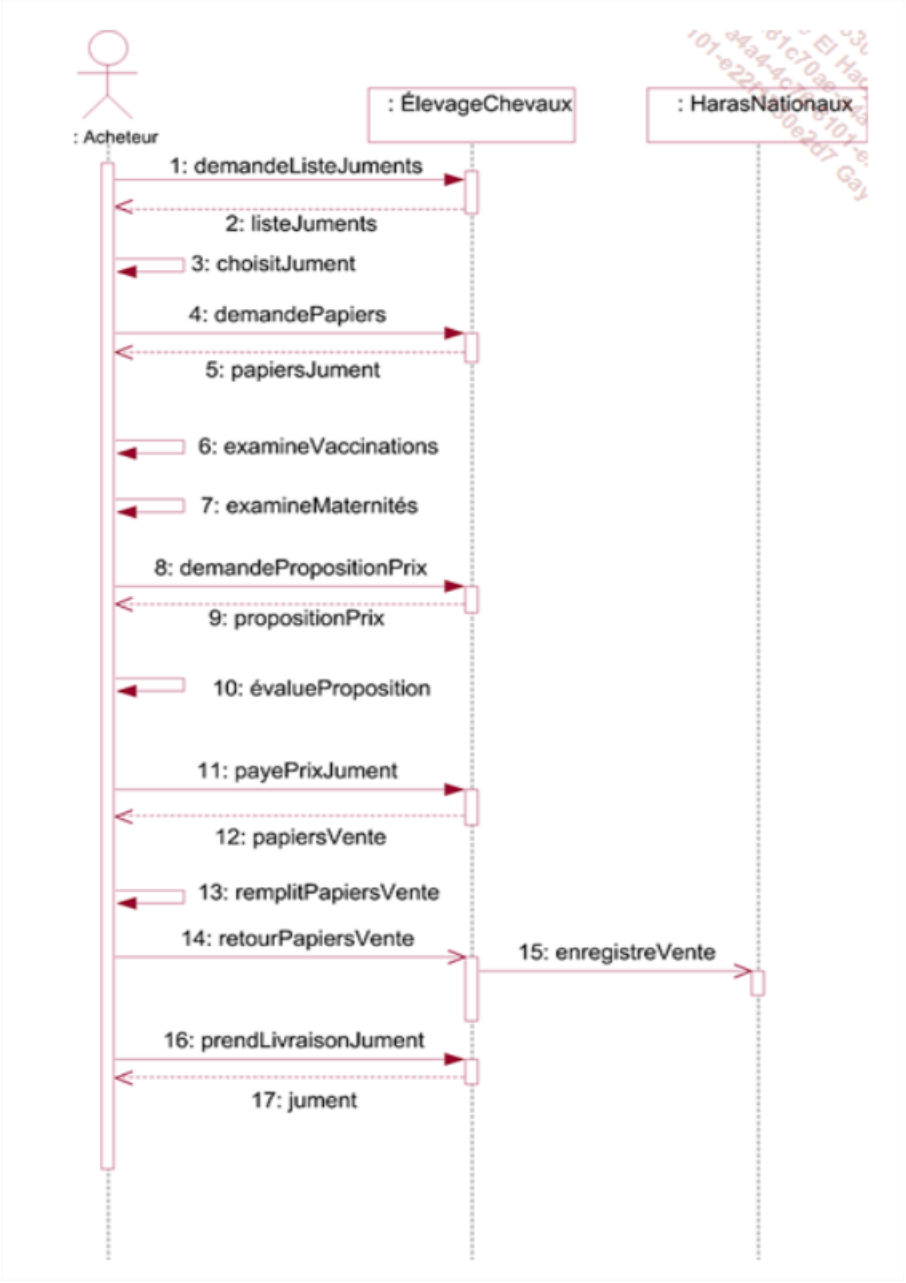
Exemple : La figure ci-dessous illustre un exemple de représentation d'objets. Les classes dont ils sont issus sont représentées au-dessus. Les différents liens qui unissent les produits et les clients montrent des représentations différentes. Le lien entre Foin et Fien ne montre que les rôles. Le lien entre Paille et Fien ne montre que le nom du lien qui n'est constitué que d'un deux-points suivi du nom de l'association. Le lien entre Paille et Laurent montre le rôle produitAcheté et le nom du lien suffixé par le nom de l'association. Enfin, le lien entre Eau et Laurent n'exhibe aucune de ses caractéristiques.



3. *Diagramme de séquence*

Le diagramme de séquence décrit la dynamique du système. À moins de modéliser un très petit système, il est difficile de représenter toute la dynamique d'un système sur un seul diagramme. Aussi la dynamique globale sera représentée par un ensemble de diagrammes de séquence, chacun étant généralement lié à une sous-fonction du système.

Le diagramme de séquence décrit les interactions entre un groupe d'objets en montrant, de façon séquentielle, les envois de message qui interviennent entre les objets. Le diagramme peut également montrer les transmissions de données échangées lors des envois de message.

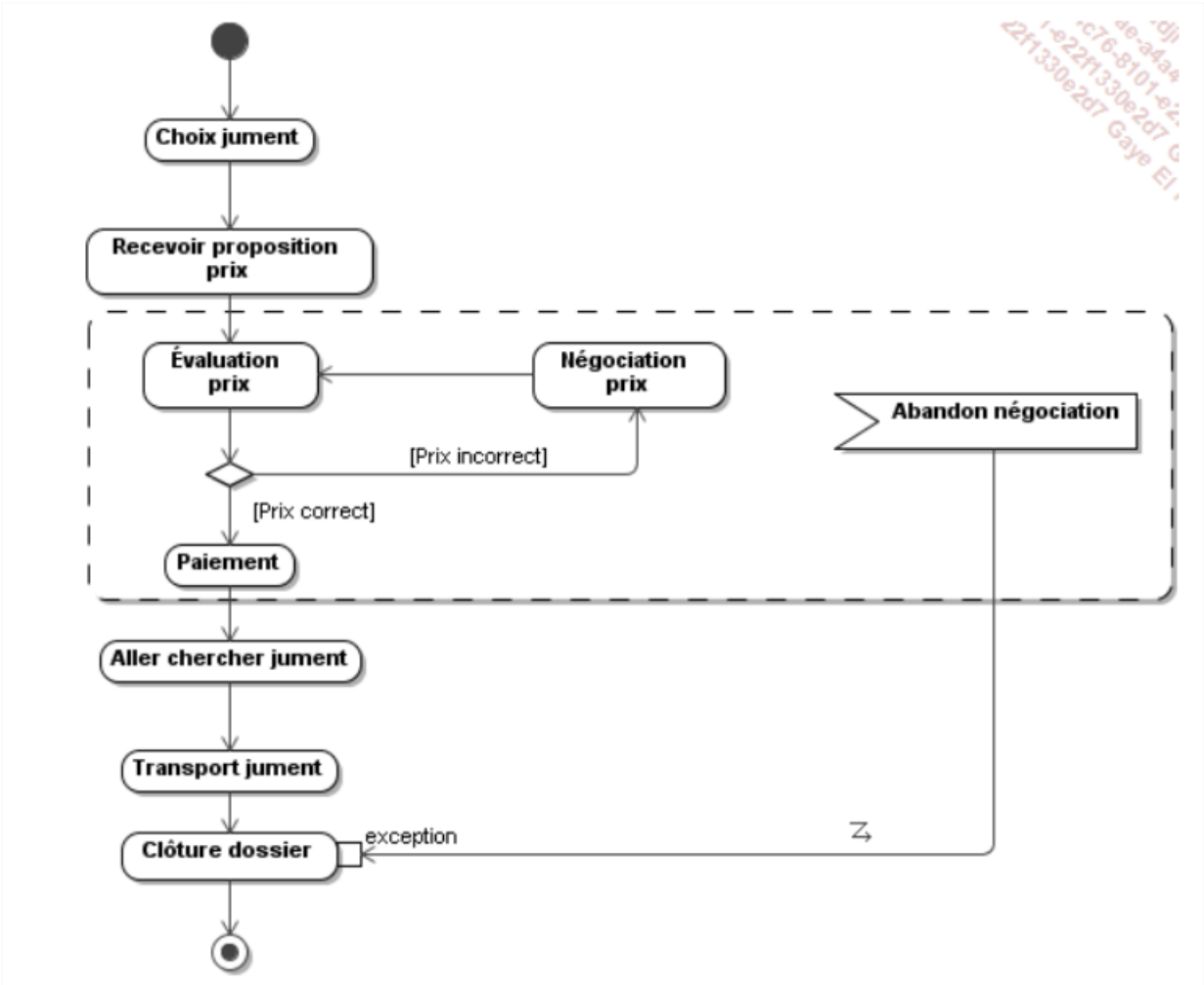


4. *Diagramme d'activité*

Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multi-threads ou multi-processus). Le diagramme d'activité est également utilisé pour décrire un flux de travail (workflow).
Un diagramme d'activité permet de modéliser un processus interactif, global ou partiel pour un système donné (logiciel, système d'information). Il est recommandable pour exprimer une dimension temporelle sur une partie du modèle, à partir de diagrammes de classes ou de cas d'utilisation, par exemple.

Le diagramme d'activité est une représentation proche de l'organigramme ; la description d'un cas d'utilisation par un diagramme d'activité correspond à sa traduction algorithmique. Une activité est l'exécution d'une partie du cas d'utilisation, elle est représentée par un rectangle aux bords arrondis.

Le diagramme d'activité est sémantiquement proche des diagrammes de communication (appelés diagramme de collaboration en UML 1), ou d'état-transitions, ces derniers offrant une vision microscopique des objets du système.

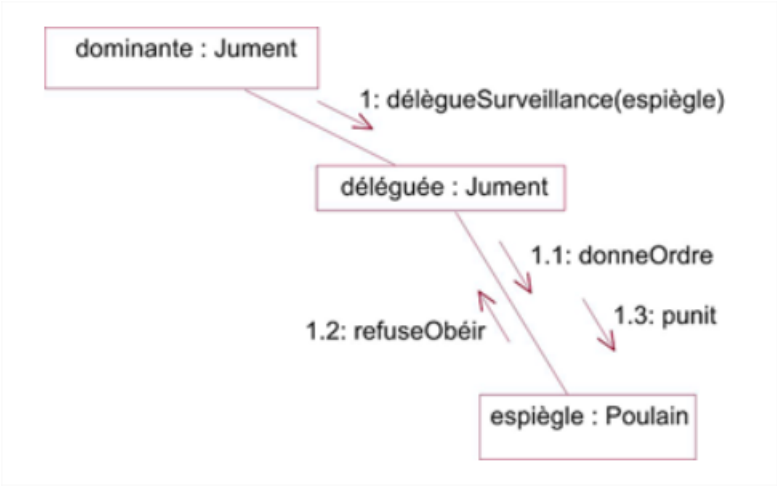


5. *Diagramme de communication*

Un diagramme de communication est un diagramme d'interactions UML 2.0 (appelé diagramme de collaboration en UML 1), représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets. En fait, le diagramme de séquence et le diagramme de communication sont deux vues différentes mais logiquement équivalentes (on peut construire l'une à partir de l'autre) d'une même chronologie, ils sont dits isomorphes.

C'est une combinaison entre le diagramme de classes, celui de séquence et celui des cas d'utilisation. Il rend compte à la fois de l'organisation des acteurs aux interactions et de la dynamique du système.

C'est un graphe dont les nœuds sont des objets et les arcs (numérotés selon la chronologie) les échanges entre objets.



6. *Diagramme paquetages (package)*

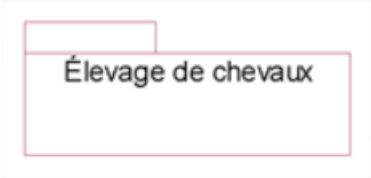
UML décrit les paquetages à l'aide d'un diagramme spécifique. Un paquetage est un regroupement d'éléments de modélisation : classes, composants, cas d'utilisation, autres paquetages, etc.

Les paquetages d'UML sont utiles lors de la modélisation de systèmes importants pour en regrouper les différents éléments. Ce regroupement structure ainsi la modélisation.

Un paquetage est représenté par un dossier. Il constitue un ensemble d'éléments de modélisation UML.

Exemple

Le paquetage regroupant les différents éléments de modélisation d'un élevage de chevaux est illustré à la figure ci-dessous.

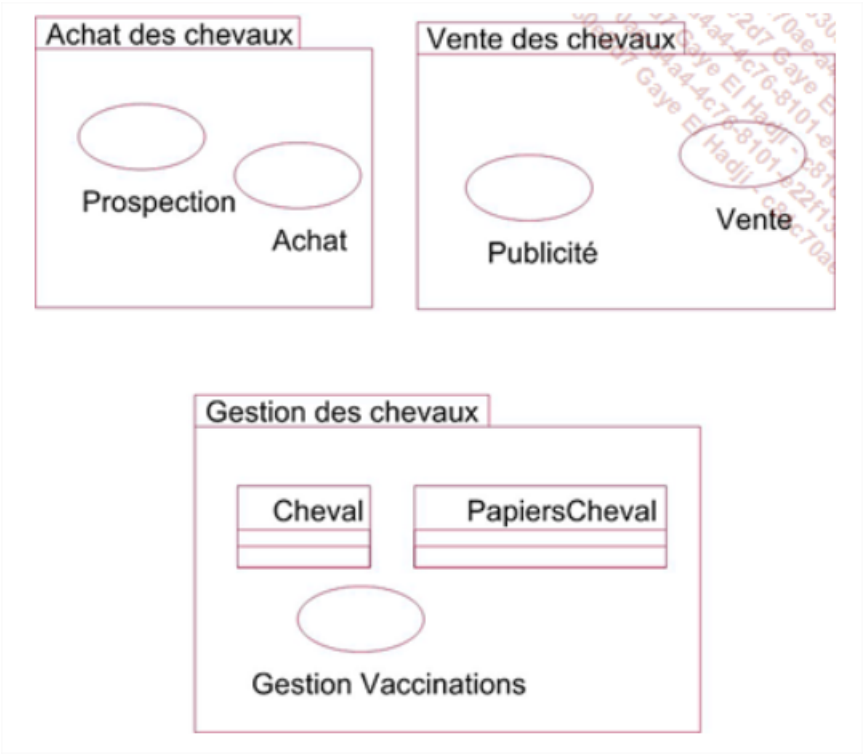


Le contenu d'un paquetage est décrit par un diagramme de paquetage. Celui-ci représente les différents éléments du paquetage avec leur propre représentation graphique. Ceux-ci peuvent être des classes, des composants, des cas d'utilisation, d'autres paquetages, etc.

Il est possible d'inclure directement les éléments d'un paquetage à l'intérieur du dossier qui le représente.

Exemple :

Le contenu du paquetage Élevage de chevaux est illustré par son diagramme. Celui-ci contient trois paquetages qui contiennent des cas d'utilisation et des classes (voir figure ci-dessous).

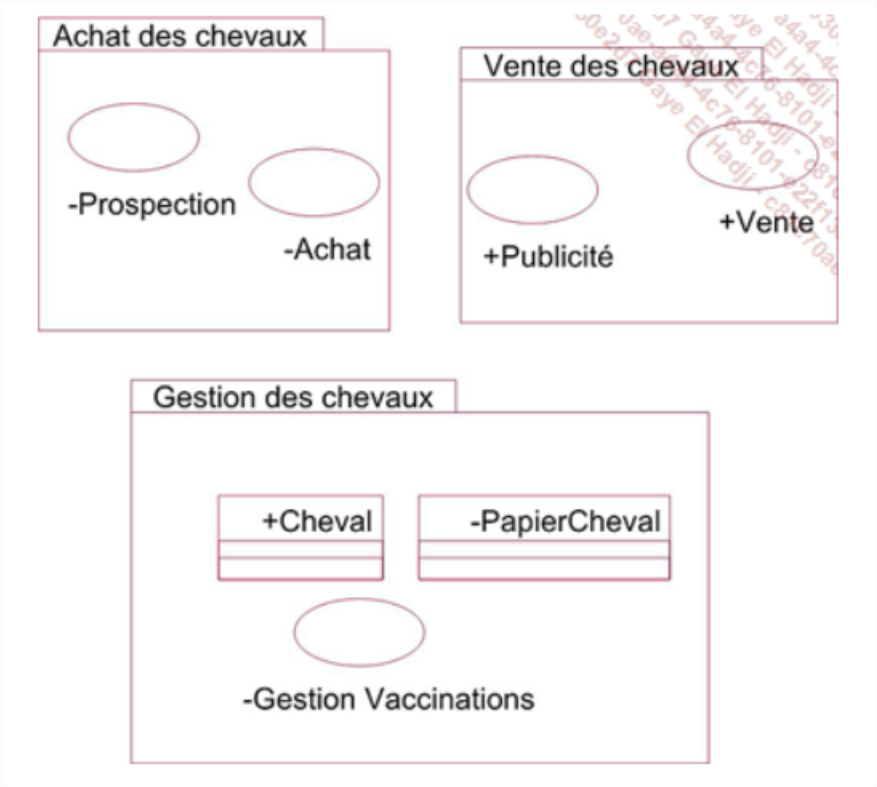


Chaque élément inclus dans un paquetage peut être accessible à l'extérieur ou encapsulé à l'intérieur de celui-ci. Par défaut, un élément est accessible à l'extérieur.

L'encapsulation est représentée par un signe plus ou un signe moins, précédant le nom de l'élément. Le signe plus signifie qu'il n'y pas d'encapsulation et que l'élément est visible en dehors du paquetage. Le signe moins signifie que l'élément est encapsulé et n'est pas visible à l'extérieur.

Exemple :

La figure ci-dessous illustre le paquetage Élevage de chevaux pour lequel l'encapsulation a été précisée.



Pour qu'un paquetage puisse exploiter les éléments d'un autre paquetage, il existe deux types de relations :

- La relation d'importation consiste à amener dans le paquetage de destination un élément du paquetage d'origine. L'élément fait alors partie des éléments visibles du paquetage de destination.
- La relation d'accès consiste à accéder, depuis le paquetage de destination, à un élément du paquetage d'origine. L'élément ne fait alors pas partie des éléments visibles du paquetage de destination.

Il n'est possible d'importer ou d'accéder à un élément que si celui-ci est défini comme visible dans le paquetage d'origine.

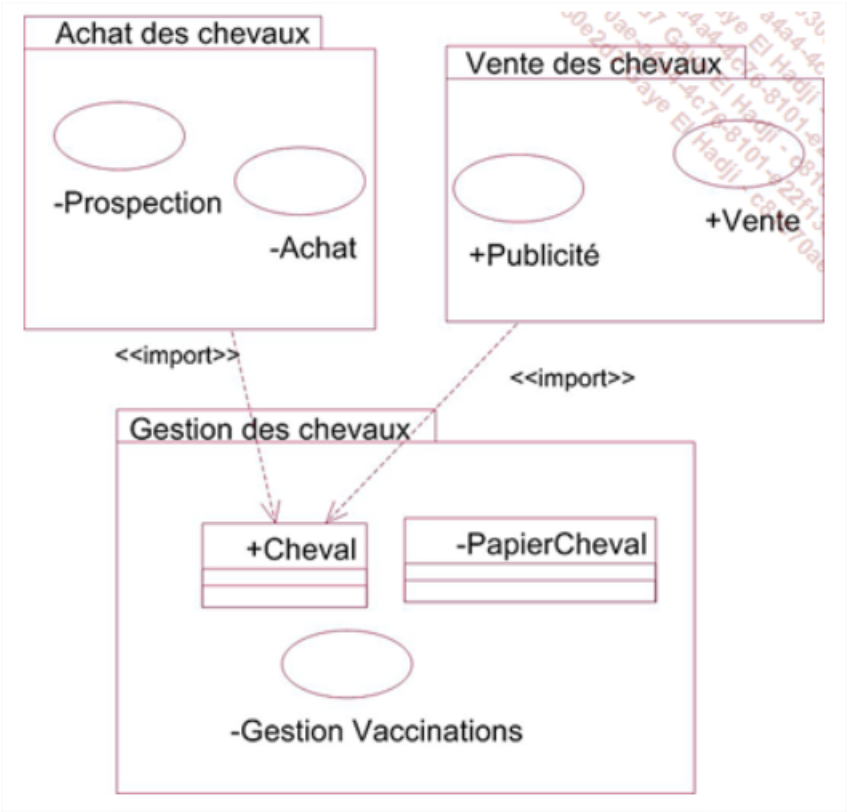
Ces deux relations peuvent également s'appliquer à un paquetage complet : elles importent ou accèdent à l'intégralité des éléments du paquetage d'origine qui sont définis comme visibles.

Ces deux relations sont des relations de dépendance qui sont spécialisées à l'aide du stéréotype «import» ou «access».

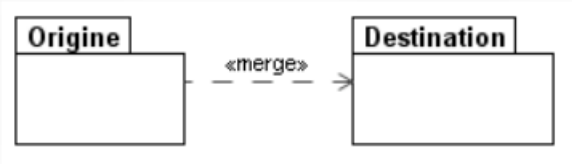
Exemple

Dans le paquetage Élevage de chevaux, les paquetages Achat de chevaux et Vente de chevaux importent la classe Cheval. Le résultat aurait été le même si ces deux paquetages avaient importé le paquetage Gestion des chevaux car la classe Cheval est la seule à être publique.

Ces deux relations d'importation sont illustrées à la figure ci-dessous.



La relation de fusion entre deux paquetages est illustrée à la figure 7.6. L'existence de cette relation engendre la fusion du contenu non privé du paquetage de destination dans le contenu du paquetage d'origine.



Les règles générales qui gouvernent la fusion entre un paquetage d'origine et un paquetage de destination sont les suivantes :

1. La relation de fusion est transformée en une relation d'importation en conservant la même orientation. Les éléments publics du paquetage de destination sont importés dans le paquetage d'origine.
2. Pour chaque élément du paquetage de destination qui peut être spécialisé (comme les classes, les cas d'utilisation, etc.) et pour lequel il n'existe pas un élément de même nom dans le paquetage d'origine, un nouvel élément de même nom est créé dans ce paquetage. Ensuite, une relation de spécialisation est introduite depuis tous les éléments du paquetage d'origine vers les éléments du paquetage de destination qui possèdent le même nom. Toutes les propriétés ainsi héritées dans les éléments du paquetage d'origine sont redéfinies. Ces propriétés redéfinies appartiennent par conséquent aux éléments du paquetage d'origine.
3. Les relations de spécialisation existantes entre les éléments du paquetage de destination sont réintroduites entre les éléments de même nom dans le paquetage d'origine.
4. Pour chaque paquetage présent dans le paquetage de destination et pour lequel il n'existe pas un paquetage de même nom dans le paquetage d'origine, un nouveau paquetage de même nom est créé dans ce paquetage. Ensuite, une relation de fusion est définie entre tous les paquetages du paquetage d'origine et les paquetages du paquetage de destination qui possèdent le même nom. Les règles de la fusion sont ainsi appliquées de façon récursive aux paquetages imbriqués.
5. Les relations d'importation et d'accès existantes entre les paquetages du paquetage de destination sont réintroduites entre les paquetages de même nom dans le paquetage d'origine.
6. Les éléments du paquetage de destination qui ne peuvent être généralisés ou spécialisés sont copiés dans le paquetage d'origine. Leurs relations avec les autres éléments dans le paquetage d'origine deviennent identiques à celles du paquetage de destination.

La fusion entre deux paquetages est introduite avec un statut de relation, et non un statut d'opération. L'application des règles de fusion détermine la description du paquetage d'origine.

Exemple :

La figure 1 ci-dessous illustre la relation de fusion. Le paquetage Jockeys est le paquetage d'origine de la relation de fusion. Il introduit des descriptions spécifiques aux cavaliers professionnels (salaire, écurie) alors que le paquetage de destination, le paquetage Cavaliers, introduit une description générale des cavaliers. Tous les éléments du paquetage Cavaliers sont publics.

La figure 2 ci-dessous illustre la description du paquetage Jockeys telle qu'elle est déterminée par l'application des règles de la relation de fusion. Les classes dont le nom est préfixé par Cavaliers:: sont les classes du package Cavaliers qui sont importées dans le paquetage Jockeys (voir règle 1). Les classes Cheval et Personne ont été introduites dans le paquetage Jockeys. Elles héritent des classes Cheval et Personne du paquetage Cavaliers dont elles redéfinissent les attributs hérités (voir règle 2). La classe Cavalier qui existait dans le paquetage Jockeys hérite maintenant de la classe Personne du même paquetage (voir règle 3) et de la classe Cavalier du paquetage Cavaliers dont elle redéfinit les attributs ancienneté et monte.

Figure 1 : Relations de fusion entre le paquetage Jockeys et le paquetage Cavaliers

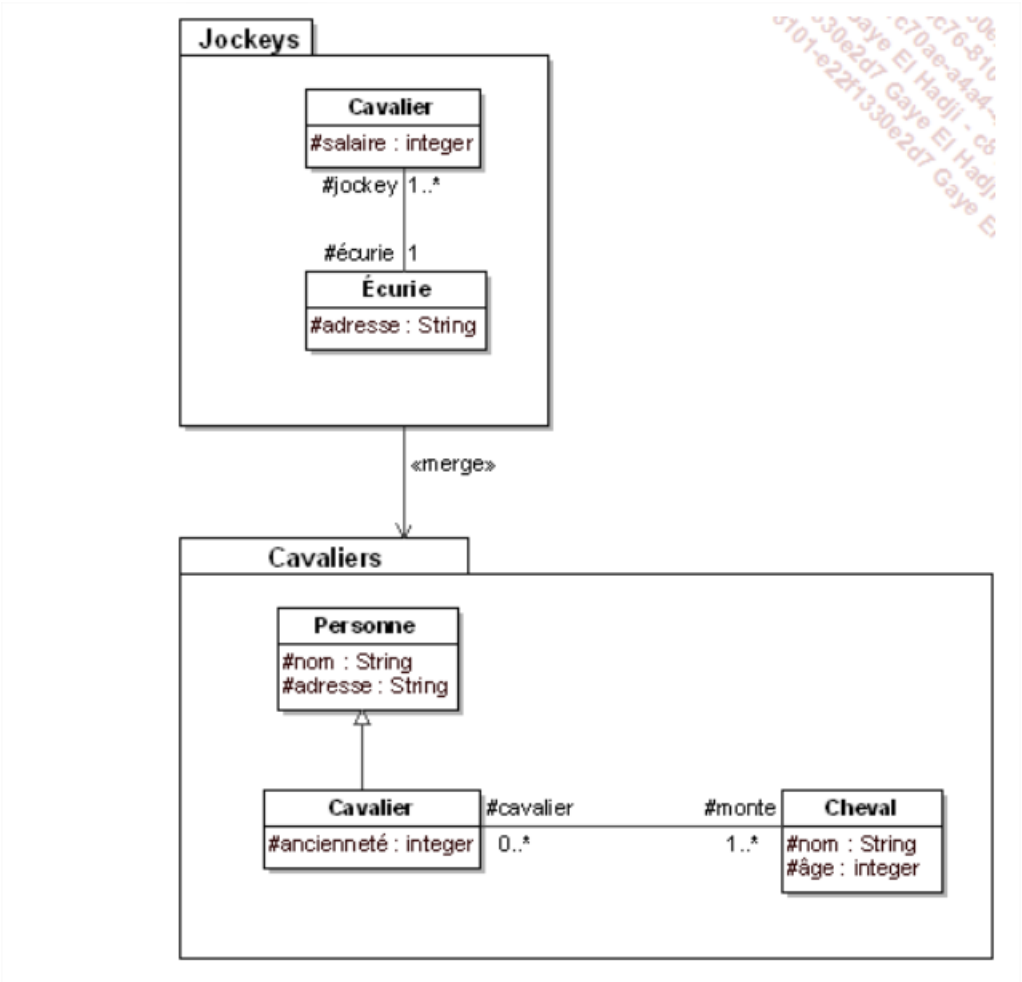
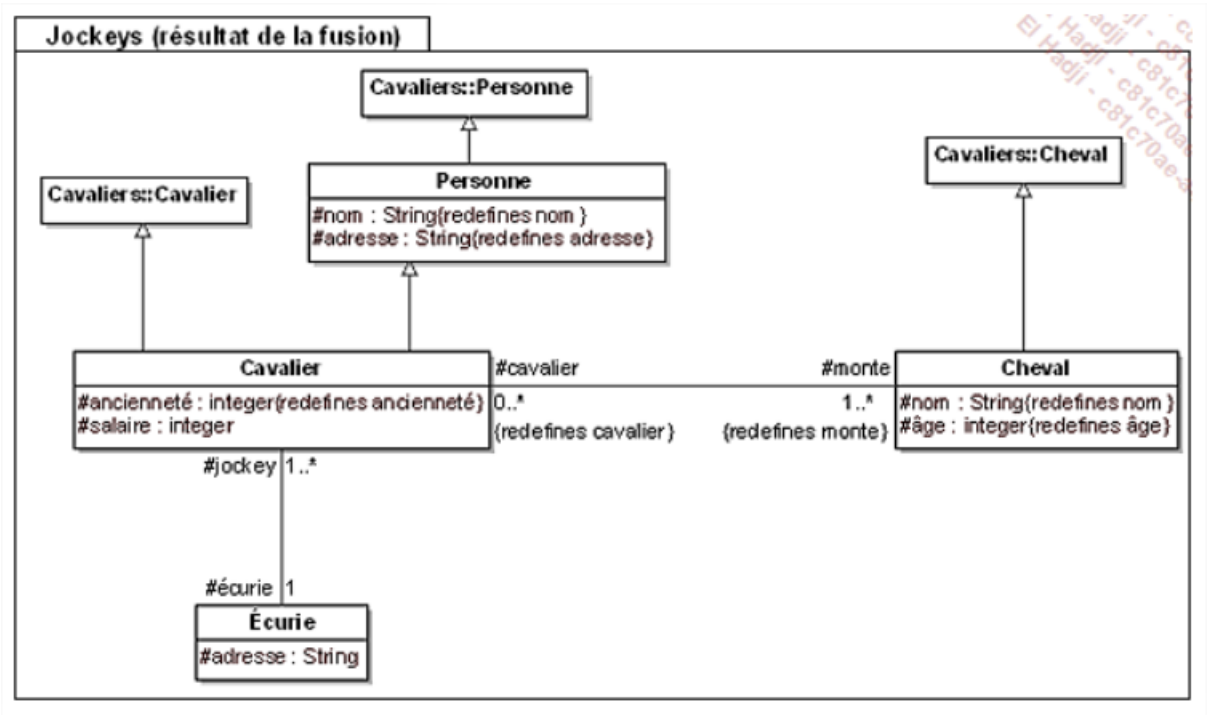


Figure 2 : Le paquetage Jockeys tel qu'il est décrit par sa relation de fusion avec le paquetage Cavaliers

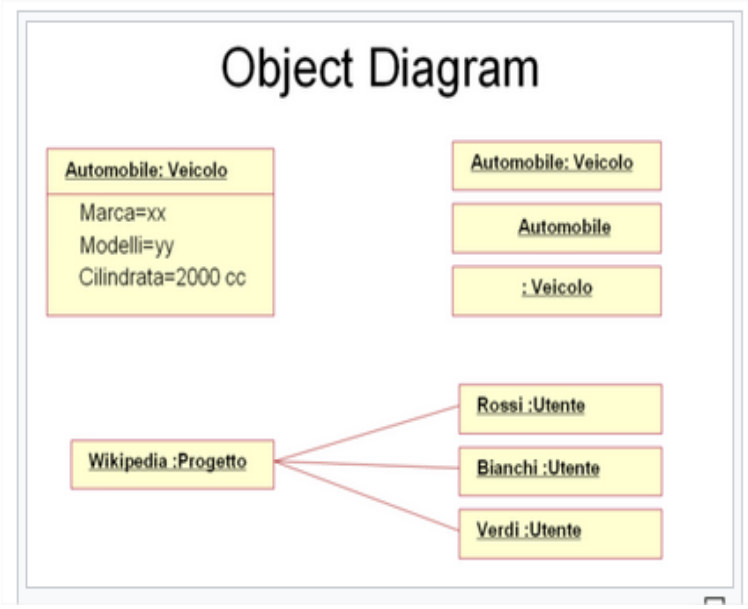


La relation de fusion entre paquetages est utilisée de façon intensive pour organiser le métamodèle d'UML. Ce métamodèle est structuré en quatre niveaux numérotés de L_0 à L_3. Le niveau L_0 contient uniquement les éléments pour modéliser les hiérarchies de classes rencontrées dans les langages de programmation par objets les plus simples. Le niveau L_3 a la capacité de décrire tout diagramme UML. Le niveau L_{i+1} est obtenu à partir du niveau L_i en associant par fusion les paquetages du niveau L_{i+1} à des paquetages du niveau L_i.

7. *Diagramme d'objets*

Le diagramme d'objets, dans le langage de modélisation de donnée UML, permet de représenter les instances des classes, c'est-à-dire des objets. Comme le diagramme de classes, il exprime les relations qui existent entre les objets, mais aussi l'état des objets, ce qui permet d'exprimer des contextes d'exécution. En ce sens, ce diagramme est moins général que le diagramme de classes.

Les diagrammes d'objets s'utilisent pour montrer l'état des instances d'objet avant et après une interaction, autrement dit c'est une photographie à un instant précis des attributs et objet existant. Il est utilisé en phase exploratoire.



8. Diagramme d'états-transitions

Le diagramme d'états-transitions représente le cycle de vie des instances d'une classe (ou une partie de ce cycle).

Il décrit les états, les transitions qui les lient et les événements qui provoquent le franchissement des transitions.

Un tel diagramme n'est utile que pour les objets qui ont un cycle de vie. D'autres objets, purement porteurs d'information, ne changent pas d'état au cours de leur vie. Pour ces objets, il est inutile de concevoir un diagramme d'états-transitions.

Etat d'un objet

L'état d'un objet correspond à un moment de son cycle de vie. Pendant qu'il se trouve dans un état, un objet peut se contenter d'attendre un signal provenant d'autres objets. Il est alors inactif. Il peut également être actif et réaliser une activité. Une activité est l'exécution d'une série de méthodes et d'interactions avec d'autres objets. Elle est liée à un objectif. Au chapitre La modélisation des activités, nous étudierons en détail leur description grâce aux diagrammes d'activités.

Exemple

Lors d'un concours de saut d'obstacles, le cheval est dans l'état de repos avant de commencer la compétition. Il s'agit d'un état où il est inactif et attend l'ordre de départ.

Lorsqu'il saute un obstacle, le cheval est dans un état où il est actif et qui se termine lorsqu'il a fini de sauter l'obstacle.

L'ensemble des états du cycle de vie d'un objet contient un état initial. Celui-ci correspond à l'état de l'objet juste après sa création. Il peut également contenir un ou plusieurs états finaux ou de terminaison. Les états de terminaison correspondent à une phase de destruction de l'objet. Il arrive également qu'il n'existe pas d'état final ou de terminaison car le cycle de vie d'un objet peut être constitué d'une boucle perpétuelle.

E**vénement**

Un événement est un fait qui déclenche le changement d'état. UML définit cinq genres d'événements :

- AnyReceivedEvent : correspond à un événement quelconque. Toutefois, il ne déclenche le changement d'état que si aucun autre événement dont le genre n'est pas AnyReceivedEvent n'enclenche au même instant un changement d'état. La syntaxe de cet événement est le mot-clé all.
- CallEvent : un événement CallEvent se produit quand la méthode spécifiée est invoquée.
- ChangeEvent : un événement ChangeEvent se produit quand un changement s'est produit dans le système. Ce changement est décrit par une expression logique qui devient vraie à l'instant où le changement se produit.
- TimeEvent : un événement TimeEvent est spécifié par une instruction indiquant un temps absolu ou relatif. Si le temps est relatif, l'instruction prend la syntaxe after expression. Si le temps est absolu, l'instruction prend la syntaxe at expression. Dans les deux cas, dès que le temps indiqué est atteint, l'événement se produit.
- SignalEvent : un événement SignalEvent se produit quand un signal est reçu.

Un signal peut être émis par tout objet, y compris par l'objet qui attend ce signal pour changer d'état.

UML propose de décrire les signaux par des classes. Chaque signal émis et reçu est alors une instance d'une classe de signaux. Pour les distinguer des autres classes, les classes de signaux sont décrites avec le stéréotype «signal». Les attributs des objets d'une telle classe sont les paramètres de message. Cette description par des classes conduit à l'organisation des signaux en hiérarchie.

UML n'impose pas de décrire précisément un événement. Dans une première phase de modélisation, les événements peuvent n'être spécifiés que par leur nom.

Transition

Une transition est un lien orienté entre deux états qui exprime le fait que l'objet a la possibilité de passer de l'état d'origine de la transition à son état de destination. Lorsque l'objet réalise ce passage de l'état d'origine à l'état de destination, la transition est alors franchi.

Une transition est généralement associée à un événement. Dans ce cas, la transition est franchie si l'objet se trouve dans l'état d'origine de la transition et si l'événement se produit. Ce franchissement a lieu que l'objet soit actif ou non. S'il est inactif, le franchissement a lieu immédiatement. S'il est actif, le franchissement a lieu dès que l'activité associée à l'état est terminée.

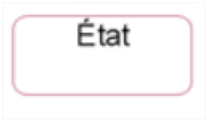
Mise en place du diagramme

Le diagramme d'états-transitions représente le cycle de vie des instances d'une classe (ou une partie de ce cycle).

Il décrit les états, les transitions qui les lient et les événements qui provoquent le franchissement des transitions.

Un tel diagramme n'est utile que pour les objets qui ont un cycle de vie. D'autres objets, purement porteurs d'information, ne changent pas d'état au cours de leur vie. Pour ces objets, il est inutile de concevoir un diagramme d'états-transitions.

Un état est représenté par un rectangle aux coins arrondis contenant son nom. La figure 8.2 montre la représentation graphique d'un état.



Dans un diagramme d'états-transitions, le premier état correspond à l'état initial de l'objet à l'issue de sa phase de création. Cet état est unique dans un diagramme d'états-transitions.

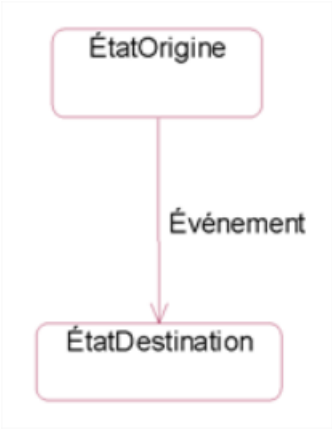
L'état initial est représenté par un point noir :

Un état final correspond à la fin du cycle de vie décrit par le diagramme d'états-transitions.

Un état final est représenté par un point noir entouré d'un cercle :

Un état de terminaison est représenté par une croix. Quand cet état est atteint, ceci signifie la fin du cycle de vie de l'objet et sa destruction.

Une transition entre deux états est représentée par un trait droit fléché reliant ces deux états (voir figure ci-dessous). L'événement qui détermine le franchissement de la transition est indiqué à proximité de la transition. Si la transition est automatique, aucun événement n'est indiqué.

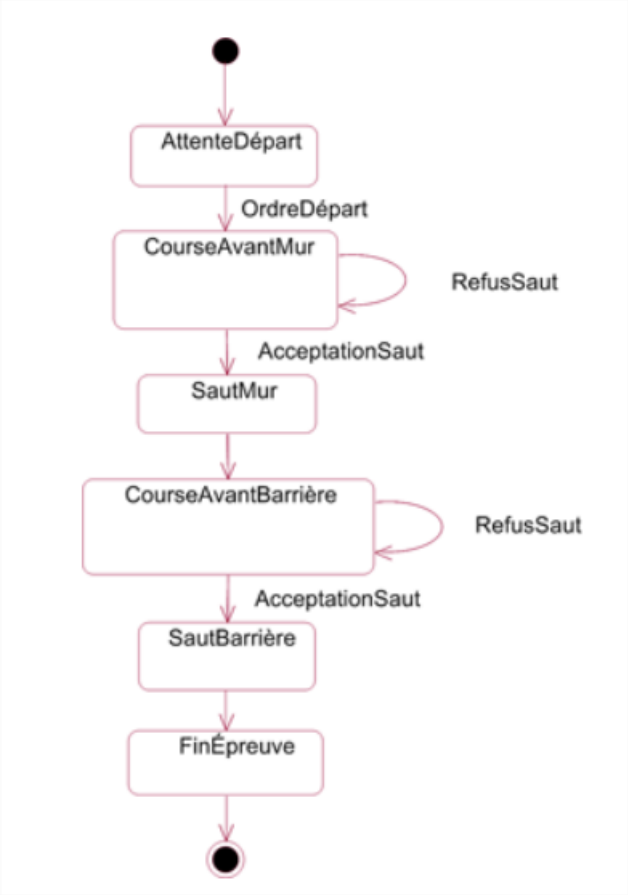


Une transition réflexive possède le même état d'origine et de destination (voir figure ci-dessous).

Exemple

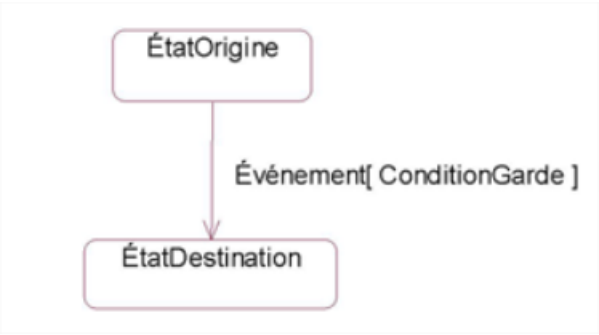
Dans un concours d'obstacles, l'épreuve consiste à demander à chaque concurrent de sauter deux ou trois obstacles différents. Il arrive que le cheval refuse de sauter un obstacle. Le concurrent peut alors recommencer le saut. La figure 8.8 représente le diagramme d'états-transitions décrivant une telle épreuve pour l'objet "concurrent de l'épreuve". Les deux obstacles sont respectivement le mur et la barrière. Ce diagramme contient des transitions réflexives et automatiques.

Le droit de sauter une nouvelle fois un obstacle est limité à deux tentatives après la tentative initiale et avant l'élimination de l'épreuve. Nous verrons par la suite comment prendre en compte cette contrainte.



Il est possible d'associer une condition à une transition, qui est alors appelée condition de garde. Pour que la transition soit franchie, il faut que la condition soit remplie en plus de la réalisation de l'événement associé, si celui-ci existe.

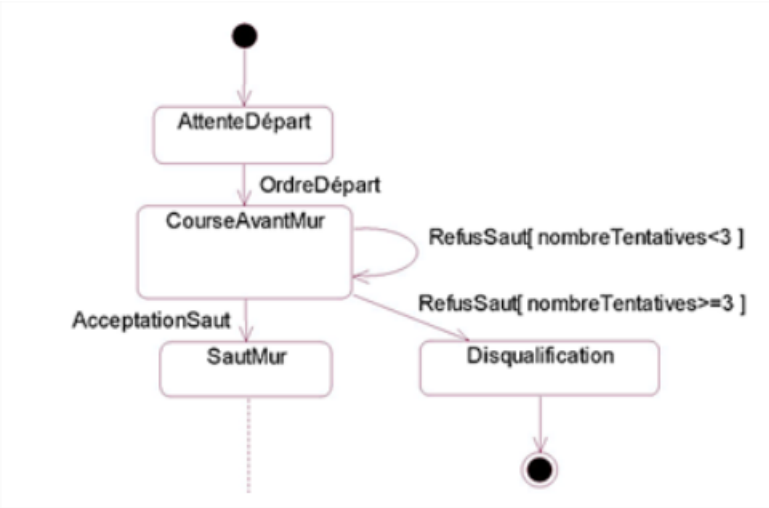
Une condition de garde est exprimée entre crochets. Si un événement est associé à la transition, la condition est exprimée à la droite du nom de l'événement (voir figure ci-dessous).



Exemple

En cas de refus de sauter un obstacle, le concurrent a le droit de recommencer deux fois. Il est donc disqualifié après la troisième tentative si elle se solde par un refus.

La figure ci-dessous illustre la prise en compte de ce nombre maximal de refus en reprenant un exemple précèdent où seul le premier obstacle est montré.



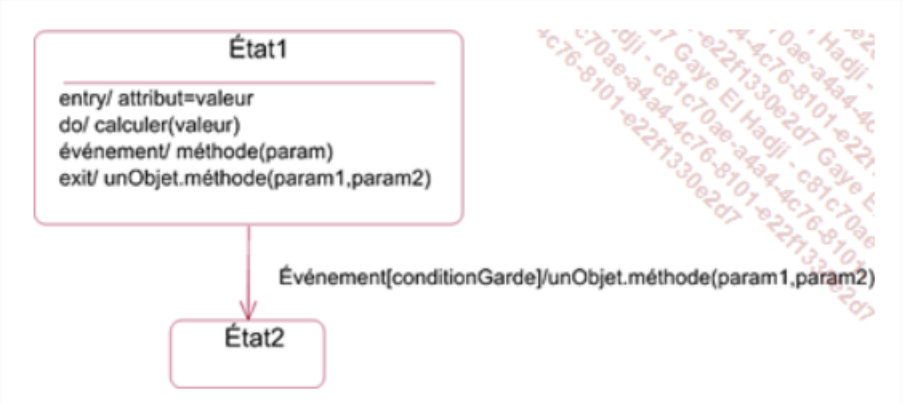
Il est possible de spécifier différentes activités :

- Pendant un état ;
- Lors du franchissement d'une transition ;
- À l'entrée et à la sortie d'un état ;
- Au sein d'un état, lors de la réception d'un événement.

Une activité est une série d'actions. Une action consiste à affecter une valeur à un attribut, à créer ou à détruire un objet, à effectuer une opération, à invoquer une méthode d'un autre objet ou "de l'objet lui-même", etc. On désignera l'autre objet par son nom comme dans les diagrammes d'interaction étudiés au chapitre La modélisation de la dynamique.

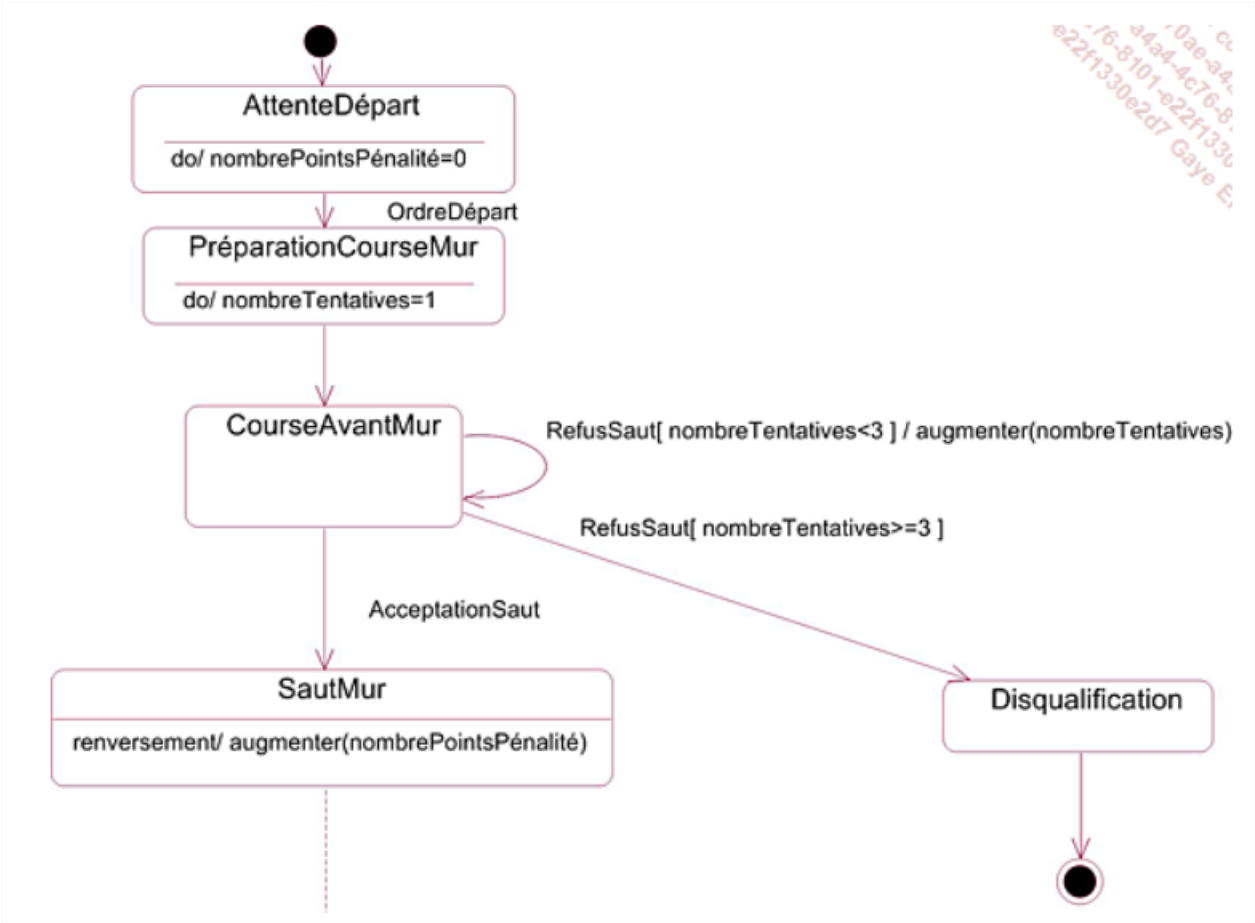
La figure ci-dessus illustre la représentation graphique de ces différentes possibilités. Une activité précédée du mot-clé entry/ est exécutée lors de l'entrée dans l'état. Le mot-clé do/ introduit l'activité réalisée pendant l'état. Une activité précédée du nom d'un événement est exécutée si cet événement est reçu. Une activité précédée du mot-clé exit/ est exécutée lors de la sortie de l'état.

Il est également possible de spécifier une activité lors du franchissement d'une transition. Dans ce cas, l'activité doit être précédée d'un /, à la suite de l'événement et de la condition de garde, s'ils existent.

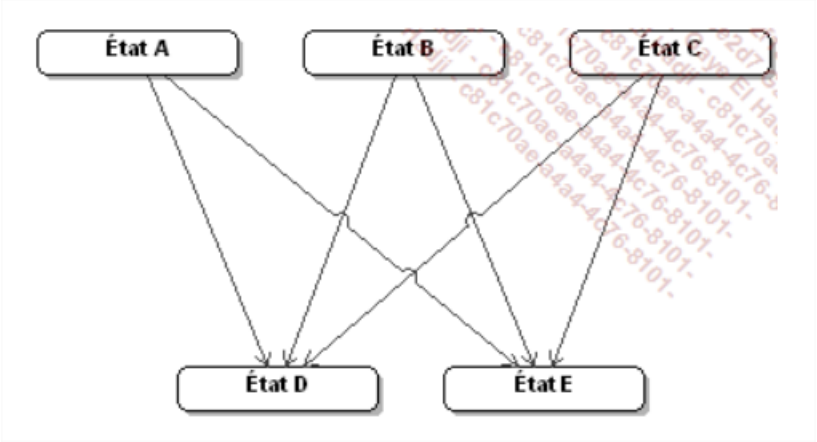


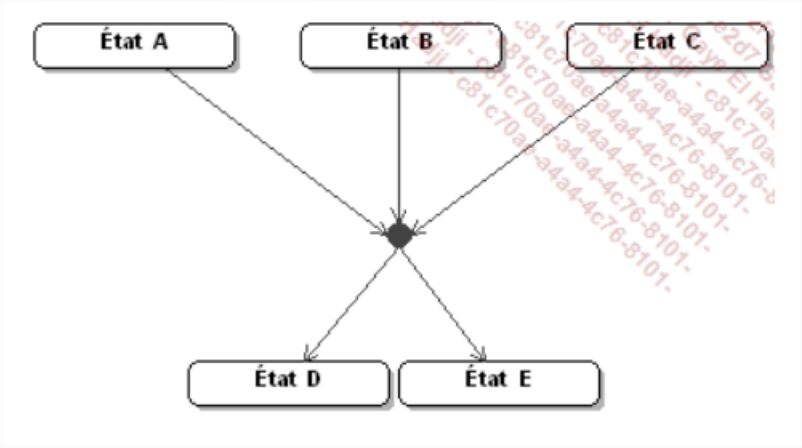
Exemple :

La figure ci-dessous illustre l'utilisation des activités au sein d'un état ou lors du franchissement d'une transition. Ceci permet notamment de gérer la valeur des attributs **nombrePointsPénalité** et **nombreTentatives** de la classe Concurrent. Le nombre de points de pénalité est augmenté si le mur est renversé, ce qui se traduit par la réception de l'événement renversement pendant l'état SautMur. Le nombre de tentatives est initialisé à 1, puis augmenté à chaque refus de sauter lors de la transition correspondante.

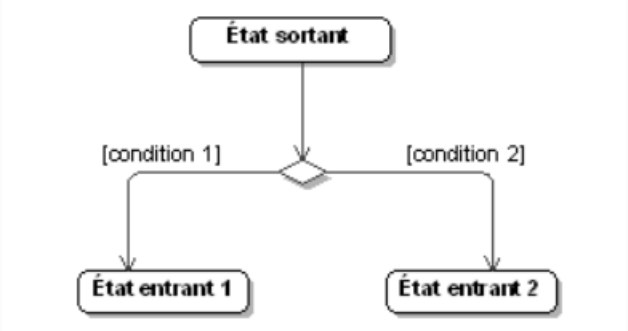


La jonction connecte plusieurs états d'origine à plusieurs états de destination, ce qui évite de décrire explicitement toutes les transitions. Les deux figures ci-dessous sont équivalentes : il y a autant de transitions d'un état d'origine à un état de destination dans les deux cas. La première figure n'utilise pas la jonction alors qu'elle est mise en œuvre à la deuxième figure.

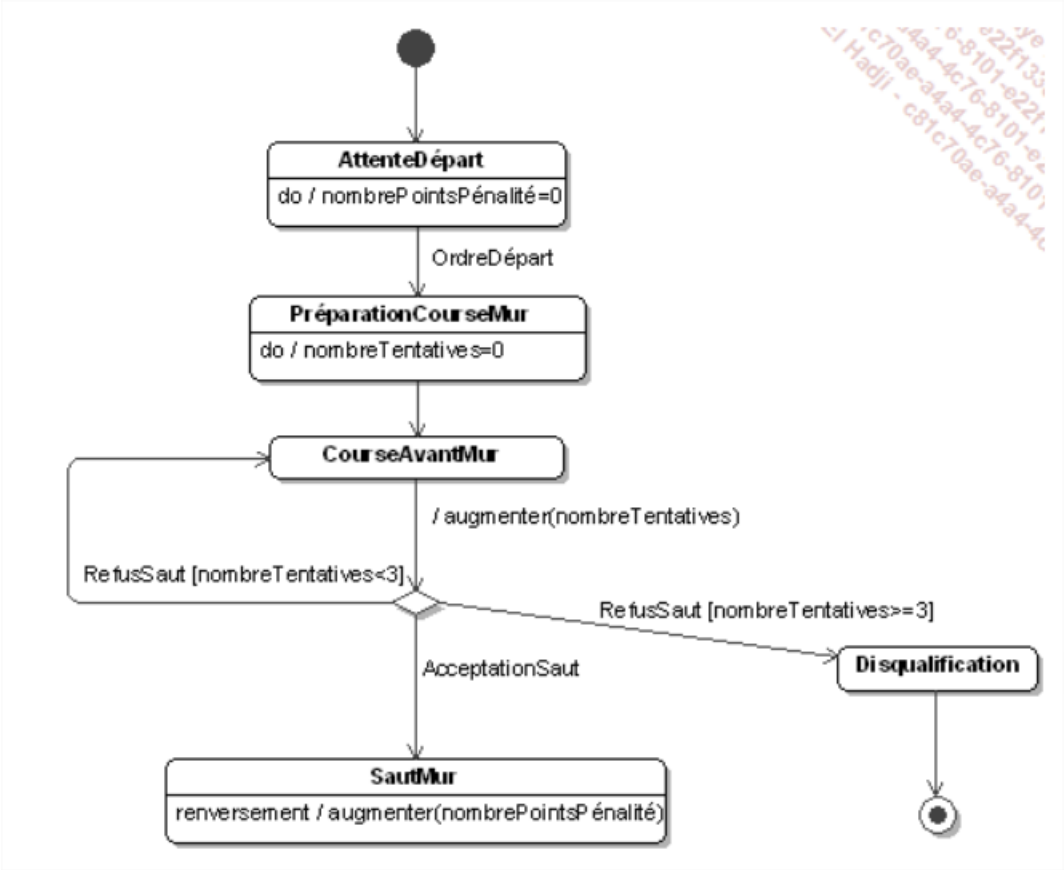


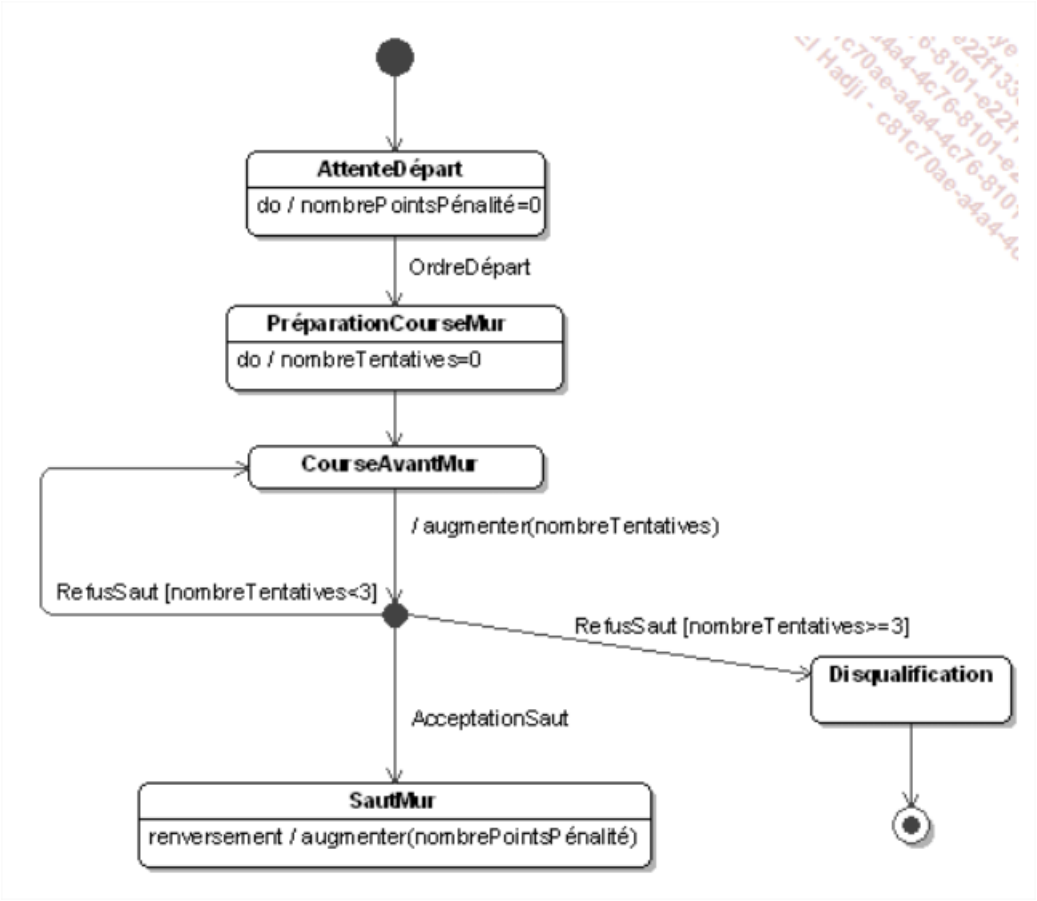


L'alternative connecte un état d'origine à plusieurs états de destination. À la différence de la jonction, l'alternative n'est équivalente qu'à une seule transition. Les sorties de l'alternative doivent être dotées de conditions de garde qui s'excluent, c'est-à-dire qui ne peuvent pas être simultanément vraies. La figure ci-dessous illustre la représentation graphique de l'alternative.



Exemples :

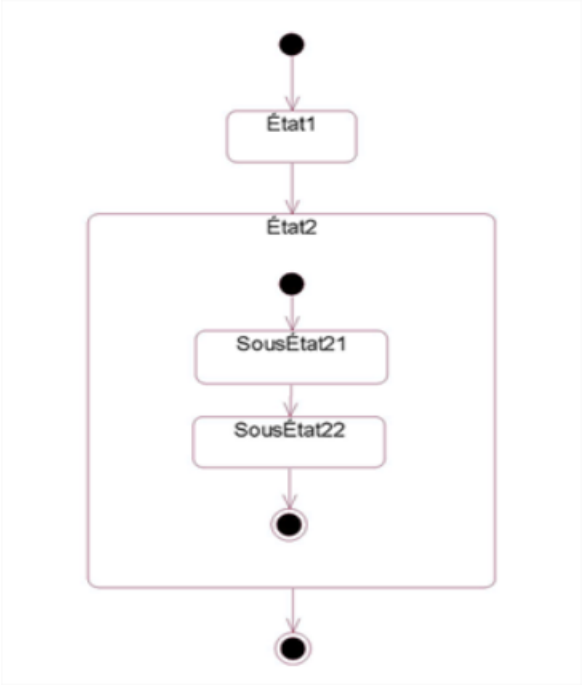

AttenteDépart
do / nombrePointsPénalité=0
OrdreDépart
PréparationCourseMur
do / nombreTentatives=0
CourseAvantMur
/ augmenter(nombreTentatives)
RefusSaut [nombreTentatives<3]
RefusSaut [nombreTentatives>=3]
Disqualification
AcceptationSaut
SautMur
renversement / augmenter(nombrePointsPénalité)


AttenteDépart
do / nombrePointsPénalité=0
OrdreDépart
PréparationCourseMur
do / nombreTentatives=0
CourseAvantMur
/ augmenter(nombreTentatives)
RefusSaut [nombreTentatives<3]
RefusSaut [nombreTentatives>=3]
Disqualification
AcceptationSaut
SautMur
renversement / augmenter(nombrePointsPénalité)

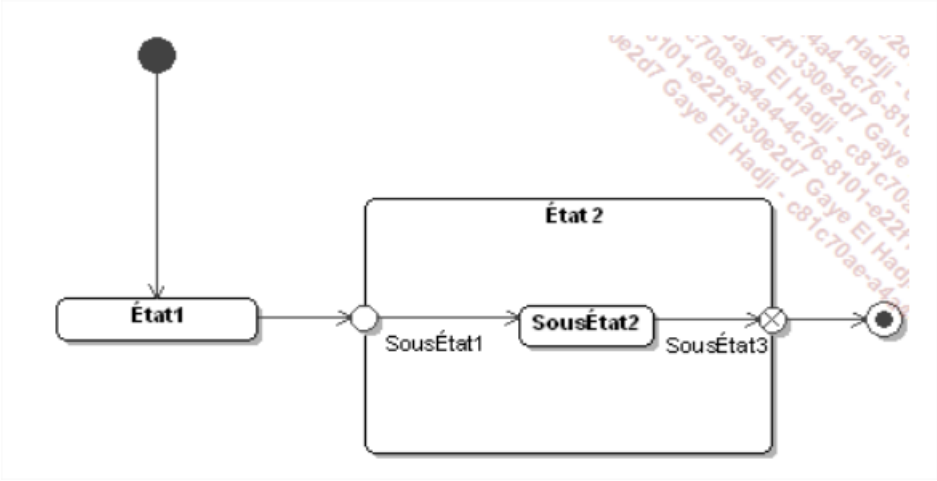
Un état peut être décrit lui-même par un diagramme d'états-transitions. Un tel état est appelé un état composé. Les états qui le composent sont appelés sous-états.

Le principe est simple : dès que l'objet passe dans l'état composé, il passe également dans le sous-état initial du diagramme interne d'états-transitions. Si l'objet franchit une transition qui fait sortir de l'état composé, il quitte également les sous-états.

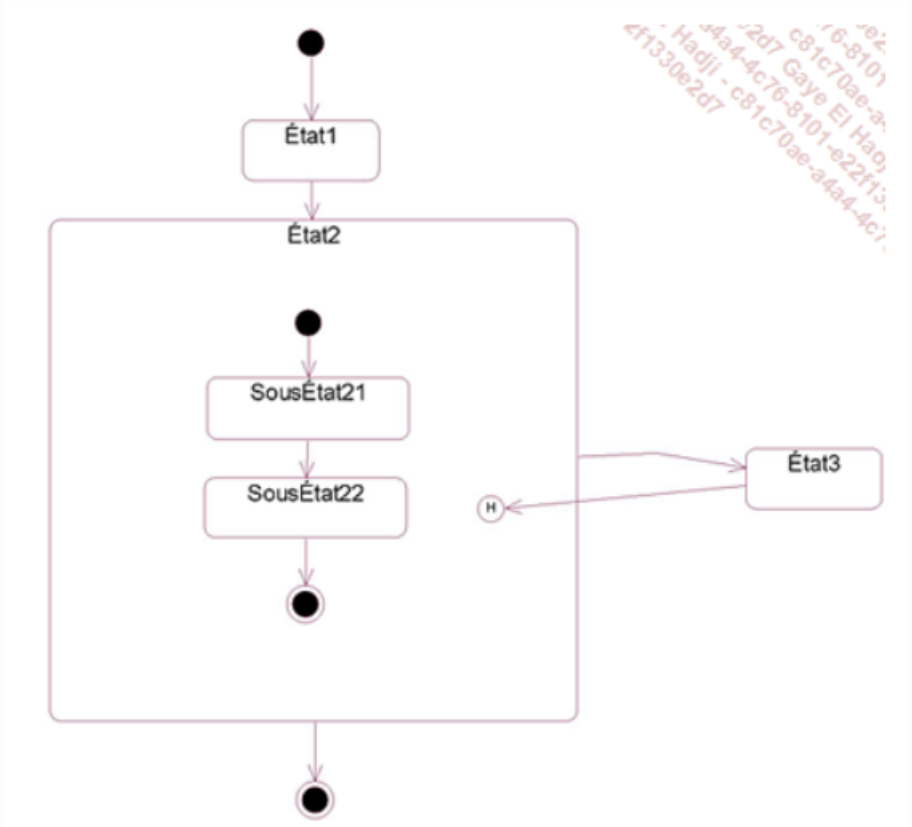
La figure ci-dessous illustre un état composé. Un diagramme interne d'états- transitions peut soit ne pas avoir d'état final, soit avoir un ou plusieurs états finaux.



La figure suivante montre une autre représentation d'un état composé avec trois sous-états. Elle est basée sur la représentation d'un point d'entrée et d'un point de sortie qui constituent des sous-états. Un point d'entrée est représenté par un simple disque. Un point de sortie est représenté par un disque contenant une croix.



Il est possible, lorsqu'un objet quitte un état composé, de mémoriser le sous-état actif pour y revenir. Pour cela, il faut utiliser le sous-état spécial de mémoire H qui représente dans l'état composé le dernier sous-état actif mémorisé. La figure suivante illustre ce sous-état spécial de mémoire.



Un sous-état peut être lui-même composé de sous-états. Dans ce cas, il existe deux sous-états de mémoire H et H*. Le premier permet de revenir au sous-état qui se trouve au niveau le plus élevé tandis que le second permet de revenir au sous-état imbriqué.

Exemple

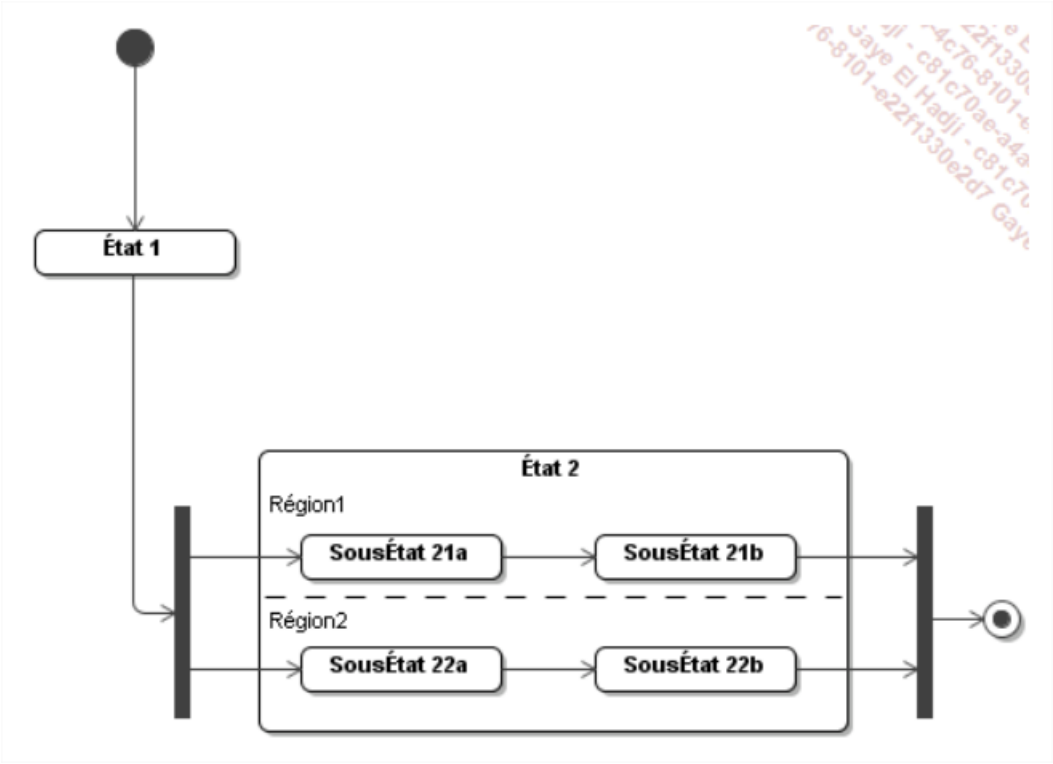
Après l'ordre de départ et jusqu'au dernier saut, un concurrent est dans l'état Concours. À tout moment, il peut être disqualifié, mais cette disqualification doit être confirmée (par exemple, en cas de contestation). Si elle est annulée, l'épreuve repart de l'état dans lequel elle s'était arrêtée.

La figure suivante illustre ce fonctionnement en utilisant un sous-état de mémoire pour revenir au dernier sous-état du concours si une disqualification est annulée.

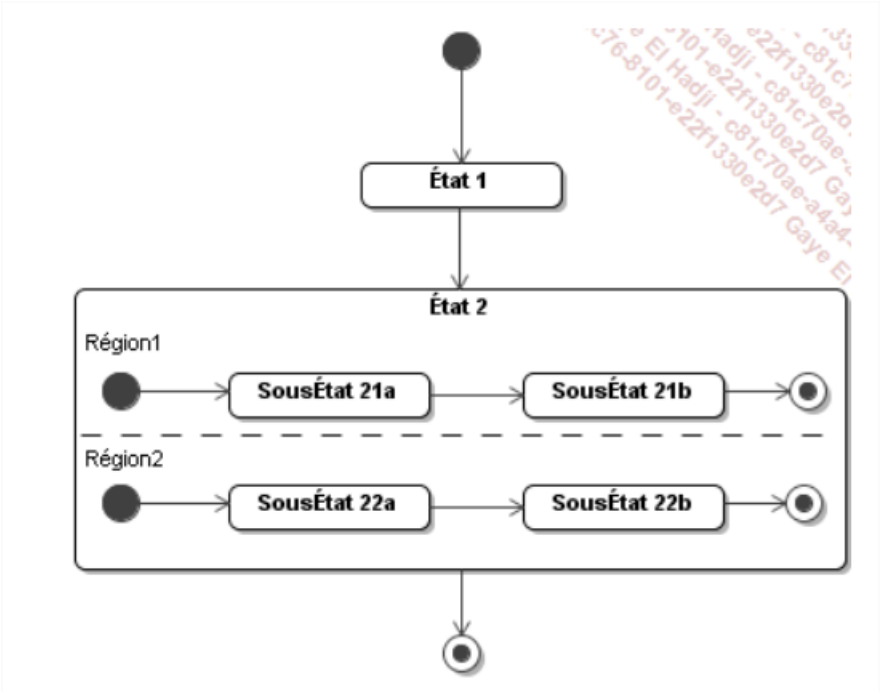
Au sein d'un objet composé, il est possible d'avoir des sous-états qui évoluent en parallèle. Pour cela, il existe une transition de type fourche qui possède plusieurs sous-états de destination. Une fois la transition franchie, l'objet se trouve dans tous les sous-états de destination.

La transition de type réunion possède plusieurs sous-états d'origine et un seul état de destination. Il faut que l'objet se trouve dans tous les sous-états d'origine pour que la transition soit franchie.

La figure suivante fournit la représentation de ces deux types de transitions. Les parties de l'état composé où les sous-états évoluent en parallèle s'appellent des régions. L'état composé contient deux régions séparées par un trait pointillé.

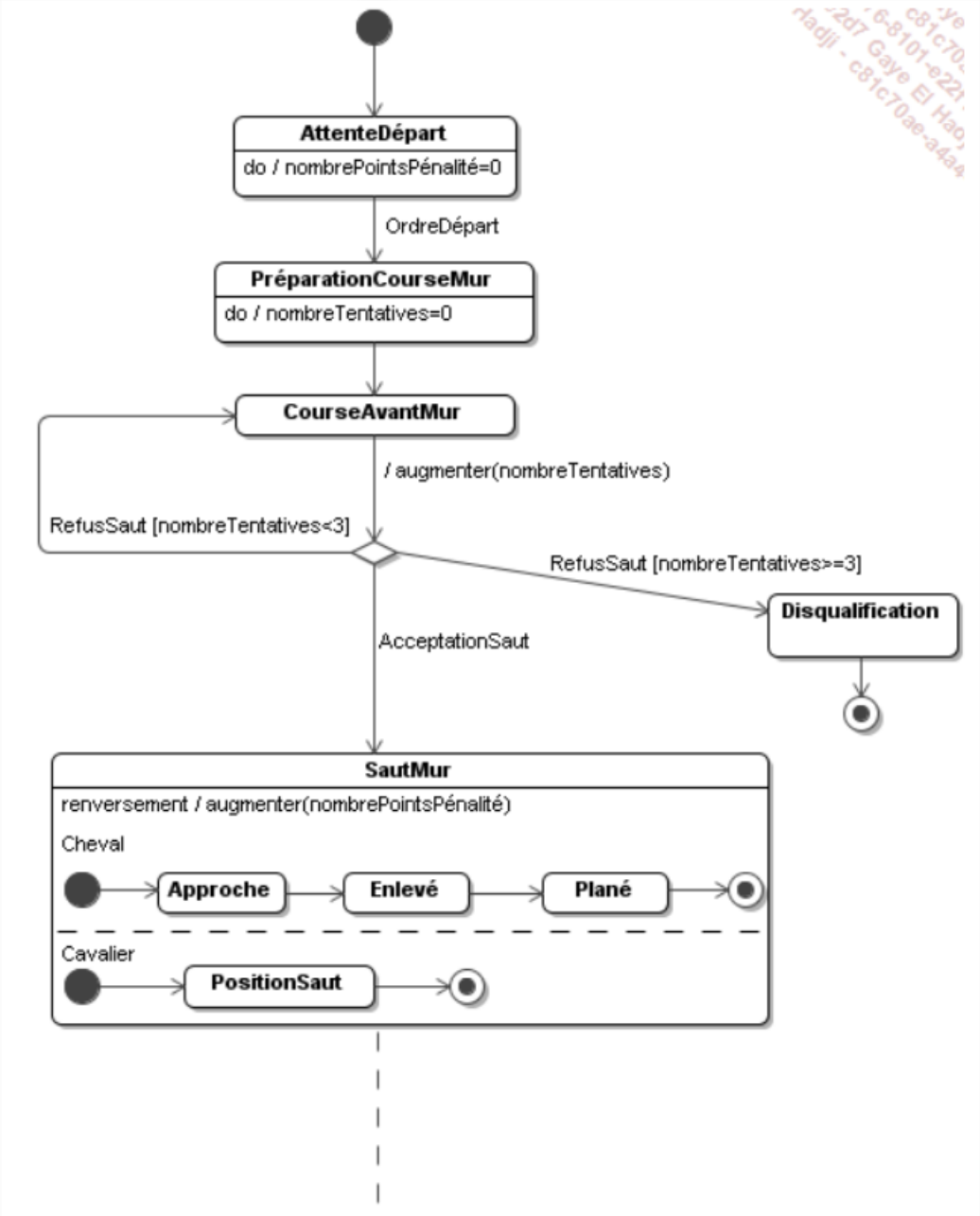


La figure suivante fournit une autre représentation du même diagramme d'états-transitions. La transition de type fourche et celle de type réunion ont été remplacées par des sous-états initiaux et des sous-états finaux. La sémantique de la figure suivante est identique à celle de la figure précédente.



Exemple :

Un saut peut être décomposé en sous-états qui sont différents pour le cheval et le cavalier, mais qui ont lieu simultanément. Ceci est illustré à la figure suivante. Dans l'état **SautMur**, les sous-états du cheval et du cavalier sont situés dans deux régions différentes afin de les distinguer. Le cheval est d'abord dans le sous-état Approche, puis il se soulève lorsqu'il se situe dans le sous-état Enlevé. Enfin, il passe dans le sous-état Plané. Le cavalier reste dans le sous-état **PositionSaut** pendant le saut.



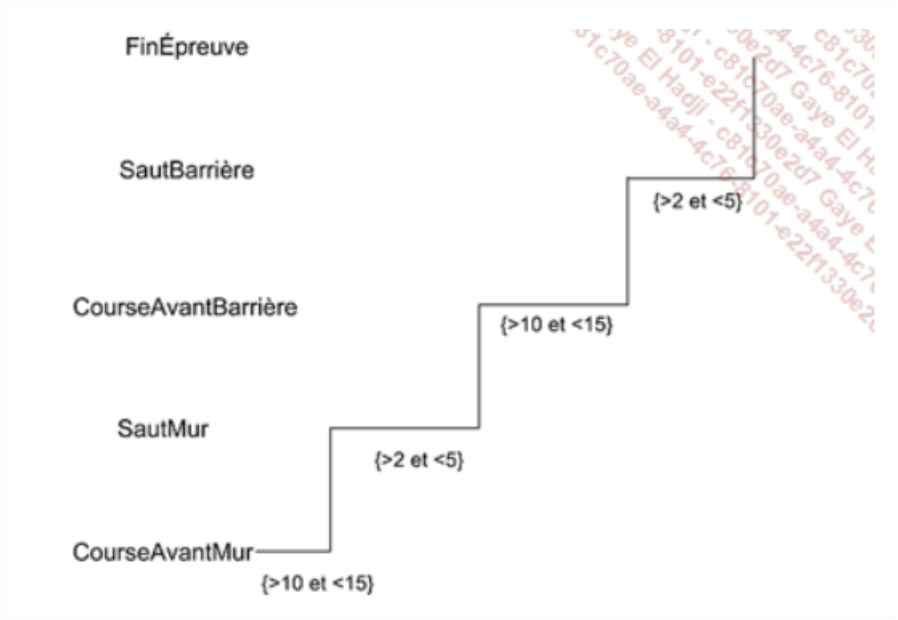
9. *Diagramme de temps (timing)*

Le diagramme de temps ou timing est introduit pour montrer les changements d'état d'un objet quand ceux-ci dépendent exclusivement du temps. Le diagramme indique alors la durée minimale et/ou maximale de chaque état à l'aide de contraintes temporelles.

La figure suivante fournit la représentation graphique du diagramme de timing.

Exemple :

Dans un concours d'obstacles, un cavalier doit réaliser l'épreuve en-deçà d'un temps maximal, sinon il est éliminé. Il décompose lui-même ce temps sur chaque partie de l'épreuve afin d'être sûr de réussir l'épreuve. La figure suivante illustre le diagramme de timing correspondant.



10. Diagramme de composants

Les composants

Un composant est une unité logicielle offrant des services au travers d'une ou de plusieurs interfaces. C'est une boîte noire dont le contenu n'intéresse pas ses clients. Il est totalement encapsulé. Cette définition d'un composant rappelle celle d'une classe implantant une interface, comme nous l'avons vu au chapitre La modélisation des objets. Une classe implantant une ou plusieurs interfaces est un composant. À l'inverse, un composant n'est pas forcément une classe. Une interface au sein d'un composant peut être implantée à l'aide de plusieurs classes ou de langages de programmation purement procéduraux comme le langage C.

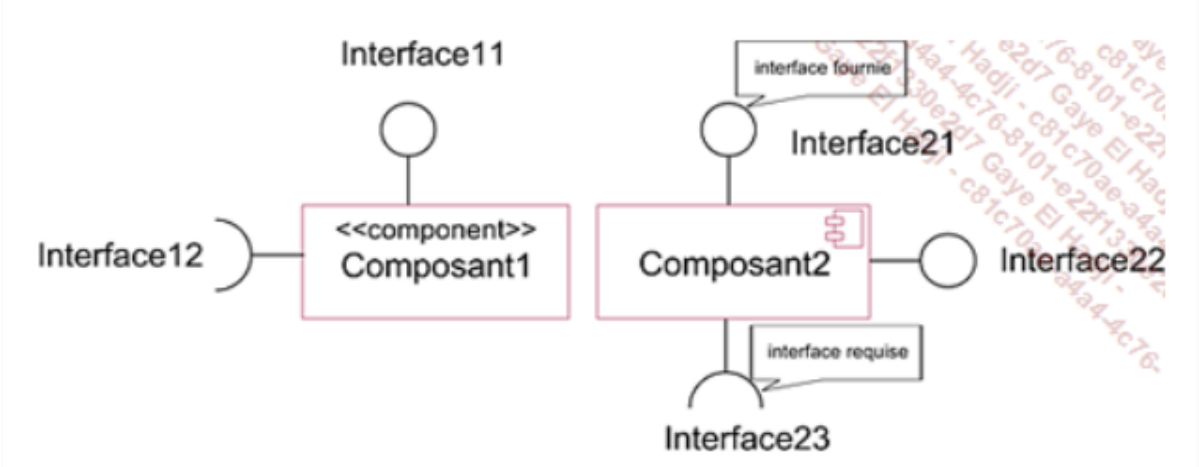
La technologie est un autre aspect des composants. Il existe aujourd'hui de nombreuses technologies de composants. Une technologie de composants définit entre autres le langage de programmation des clients, l'environnement d'exécution, l'intégration à la plateforme logicielle sous-jacente (Windows, Java, etc.). Un composant qui utilise une technologie bénéficie d'un standard et devient, par conséquent, commercialisable : il est disponible sur étagère.

Il existe plusieurs technologies de composants :

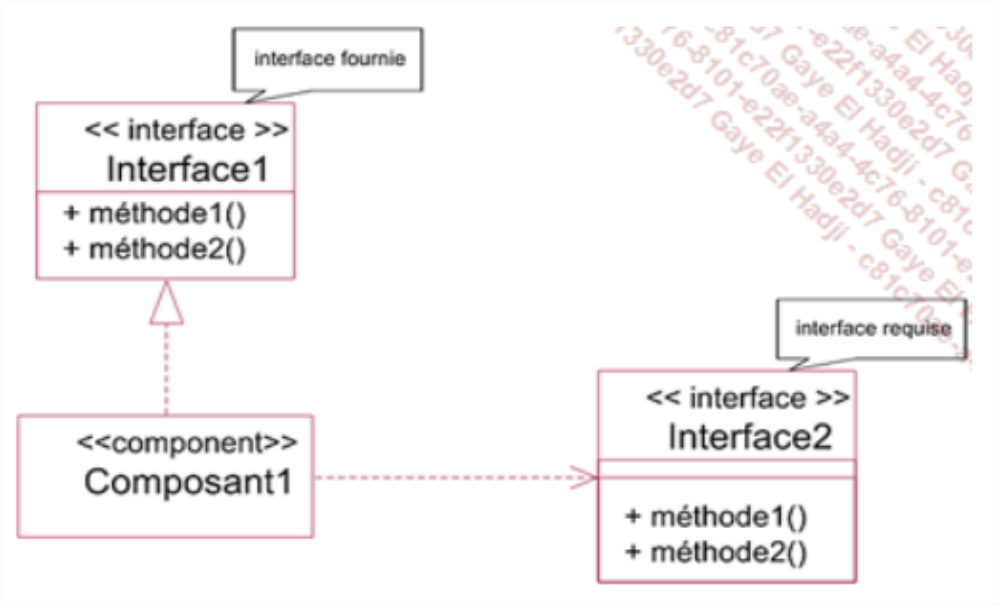
- Les composants COM et .NET ;
- Les composants Java : JavaBeans et Enterprise JavaBeans.

Un composant peut dépendre d'autres composants pour réaliser ses opérations internes. Il est alors le client de ces autres composants. Comme tout client d'un composant, il n'en connaît pas la structure interne. Il ne dépend donc que de la ou des interfaces des composants dont il est client. Ces interfaces sont appelées les interfaces requises du composant client. Les interfaces qui décrivent les services offerts par un composant sont appelées les interfaces fournies.

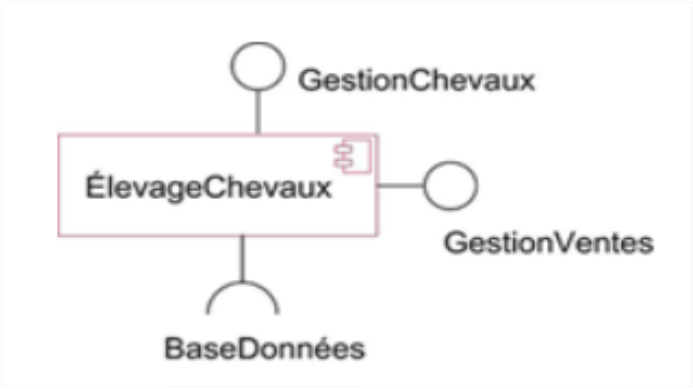
La notation graphique d'une interface fournie est identique à celle que nous avons présentée au chapitre La modélisation des objets. Une interface requise est représentée par un demi-cercle. Un composant est représenté dans un rectangle avec le stéréotype «component». Ce stéréotype peut être remplacé par l'icône du composant. La figure ci-dessous illustre la représentation graphique d'un composant sous deux formes, l'une avec le stéréotype, l'autre avec l'icône.



Il est également possible d'utiliser la relation de réalisation pour les interfaces fournies et la relation de dépendance pour les interfaces requises. La figure ci-dessous illustre cette représentation alternative. Celle-ci présente l'avantage de détailler les signatures de méthodes contenues dans les interfaces.



Exemple : Un composant de gestion d'un élevage de chevaux fournit une interface pour la gestion des chevaux et une interface pour la gestion des ventes. Par ailleurs, il requiert un composant de base de données. La figure ci-dessous illustre ce composant.

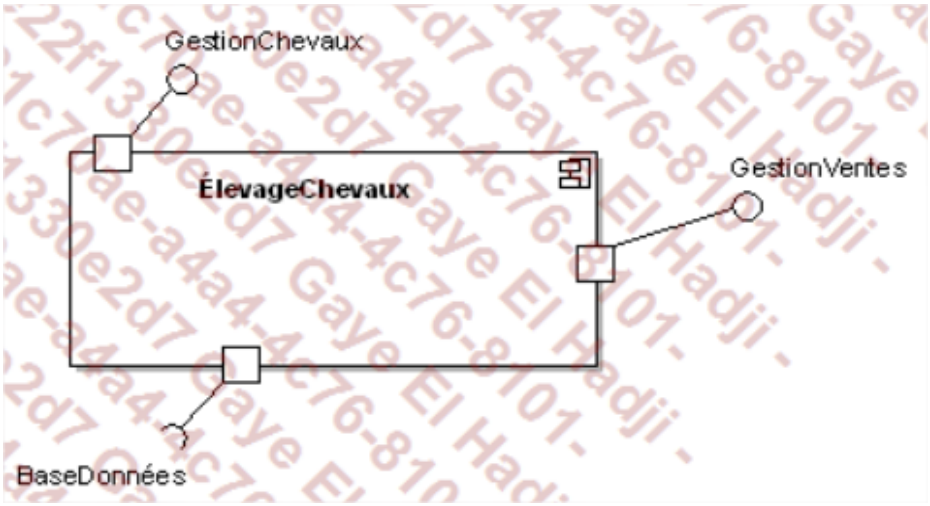


Les ports

Les composants peuvent intégrer des ports qui sont des points d'interaction avec les objets externes ou, de façon interne, entre les parties du composant.

Chaque port possède une ou plusieurs interfaces fournies ou requises. Celles-ci définissent les interactions possibles du port. Chaque interface peut être soit une interface fournie soit une interface requise.

Exemple : La figure ci-dessous illustre des composants où les interactions entre ce composant et les objets externes sont gérées par des ports.



Les stéréotypes des composants

Nous savons qu'un stéréotype est utilisé pour spécialiser un élément. Vous trouverez à la suite une liste des principaux stéréotypes de composant.

«specification» : ce stéréotype indique qu'il s'agit d'un composant qui spécifie seulement des interfaces fournies et requises.

«implement» : ce stéréotype indique qu'il s'agit d'un composant qui ne spécifie aucune interface, mais introduit l'implantation d'un autre composant dont le stéréotype est «specification».

«entity» : le composant mémorise des informations persistantes.

«process» : l'exécution du composant est basée sur un processus, un thread ou des transactions.

«subsystem» : le composant implante une partie d'un système plus important, qui peut être également décrit par un composant.

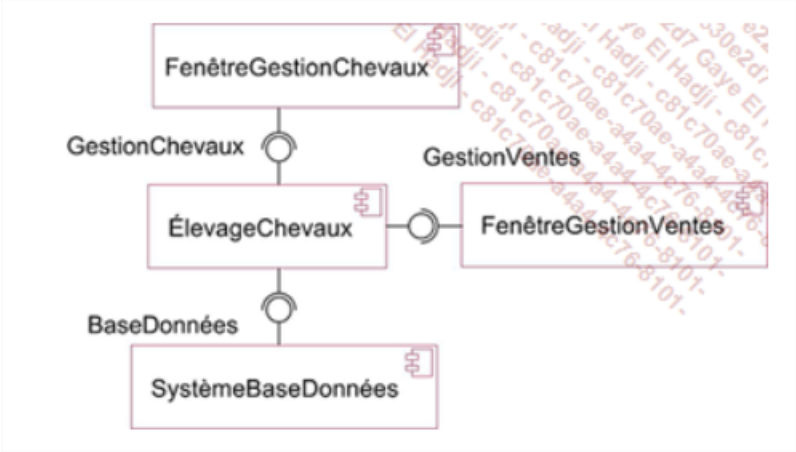
«service» : le composant a une fonctionnalité purement fonctionnelle, sans état.

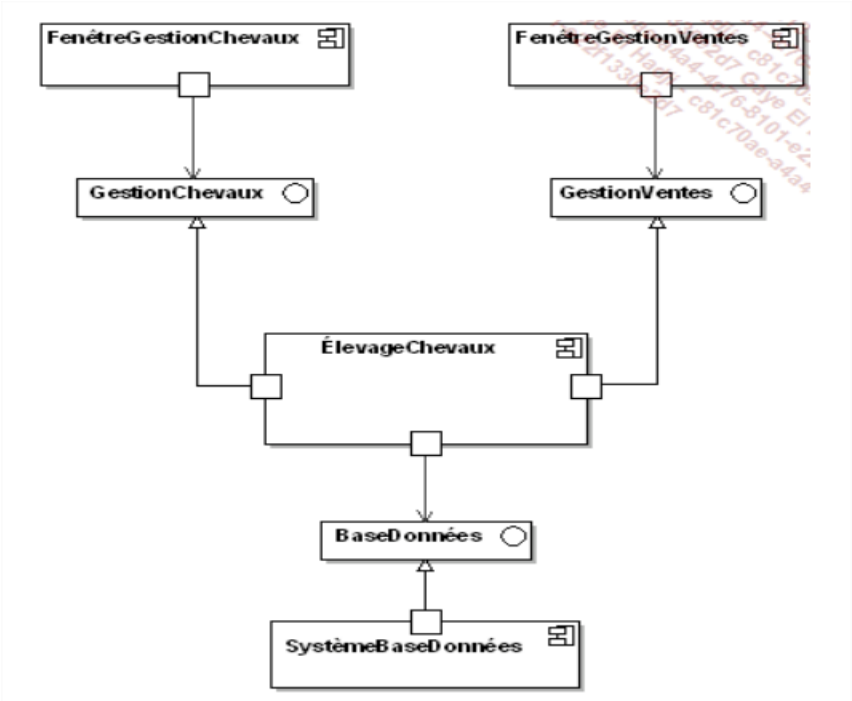
Architecture logicielle par composants

Dans l'approche par objets, l'architecture logicielle d'un système est construite par un assemblage de composants liés par des interfaces fournies et des interfaces requises. Le diagramme de composants décrit cette architecture.Les connecteurs qui relient les interfaces requises et les interfaces fournies s'appellent des connecteurs d'assemblage.

Exemple : Le système d'information d'un élevage de chevaux est réalisé par assemblage de composants. La première figure ci-dessous illustre le diagramme de composants de ce système. Les composants FenêtreGestionChevaux et FenêtreGestionVentes gèrent respectivement à l'écran une fenêtre consacrée à la gestion des chevaux et une fenêtre dédiée à la gestion des ventes de chevaux. Le composant SystèmeBaseDonnées est un système de base de données.

La deuxième figure ci-dessous illustre le même diagramme avec les différents ports des composants.



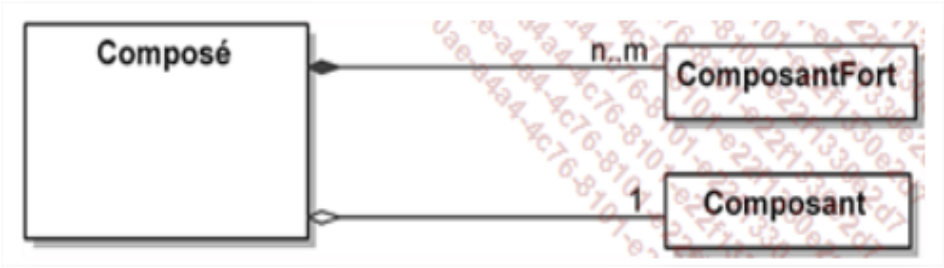


11. Diagramme de structure composite

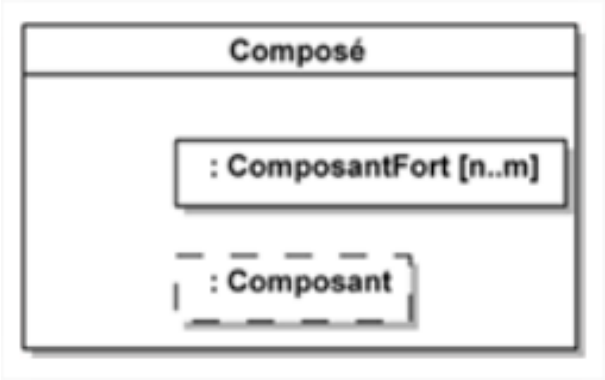
Le diagramme de structure composite a pour premier objectif de décrire précisément un objet composé. Un tel diagramme n'a pas vocation à remplacer le diagramme de classes mais à le compléter.

Dans le diagramme de structure composite, l'objet composé est décrit par un classificateur tandis que ses composants sont décrits par des parties. Un classificateur et une partie sont associés à une classe, dont la description complète est réalisée dans un diagramme de classes.

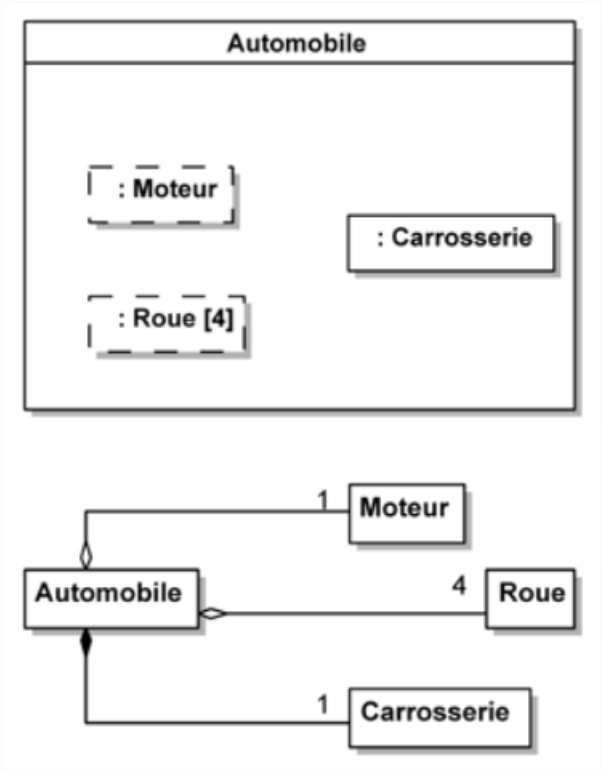
Considérons l'objet composé décrit par le diagramme de classes de la figure ci-dessous. Il possède un composant issu d'une composition forte et un autre issu d'une agrégation.



La figure ci-dessous montre le diagramme de structure composite correspondant à cet objet. Les composants sont intégrés au sein du classificateur qui décrit l'objet composé. Les parties possèdent un type qui est la classe du composant. La cardinalité est indiquée entre crochets. Par défaut, elle vaut 1. Un composant issu d'une agrégation est représenté par une ligne en pointillés, un composant issu d'une composition forte est représenté par une ligne continue.

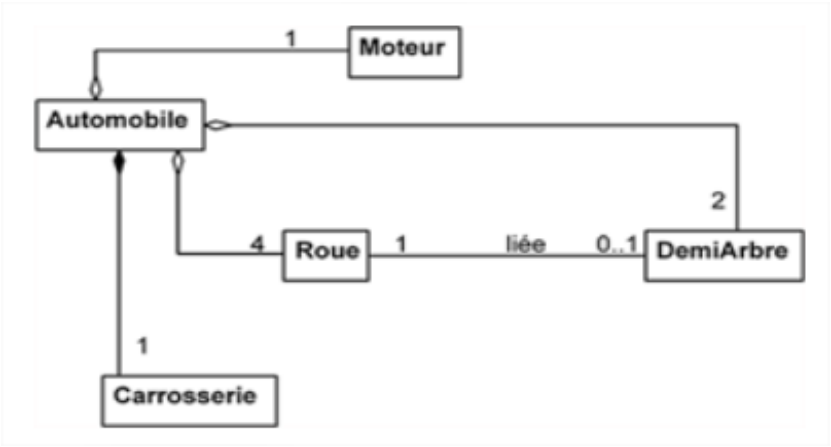


Exemple : La figure ci-dessous illustre un exemple de diagramme de structure composite décrivant une automobile en tant qu'objet composé. Le diagramme de classes correspondant est représenté en dessous.



Remarque : Dans le diagramme de structure composite, Moteur, Carrosserie et Roue ne sont pas des classes mais des parties. Une partie est toujours prise en compte au sein d'un classificateur.

Le diagramme de classes de la figure 6.64 illustre à nouveau une automobile en tant qu'objet composé. Il introduit l'association liée entre les roues et les demi-arbres qui assurent la transmission entre le moteur et les roues avant, qui sont les roues motrices.



La cardinalité de l'association liée est 0..1 à l'extrémité du demi-arbre. En effet, la cardinalité vaut 1 pour les roues avant et vaut 0 pour les roues arrière. Cette dernière information ne peut pas être décrite dans le diagramme de classes, à moins d'introduire deux sous-classes de Roue : **RoueAvant** et **RoueArrière**. Cependant, introduire deux sous-classes pour préciser une cardinalité présente l'inconvénient d'alourdir le diagramme de classes. Cette possibilité n'est pas souhaitable.

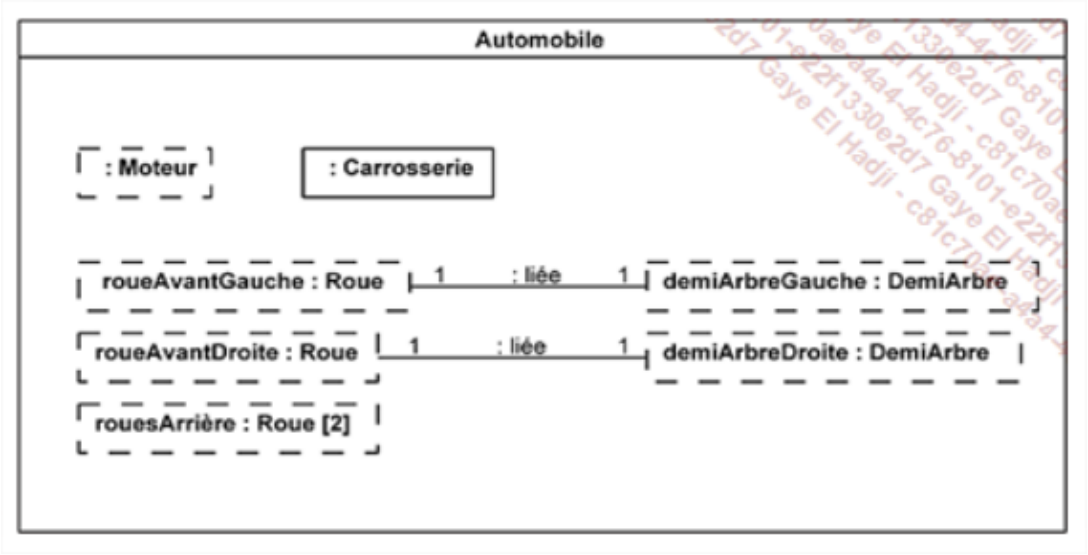
Le diagramme de structure composite permet de spécifier le rôle d'une partie. Le rôle décrit l'utilisation de la partie au sein de l'objet composé. La figure 6.65 introduit trois parties pour les roues correspondant à la roue avant gauche, la roue avant droite et les deux roues arrière. La cardinalité des parties est adaptée en conséquence. Le nom du rôle est indiqué avant celui du type dans la partie.

Remarque : Cette notation n'est pas la même que celle utilisée dans le diagramme d'objets. En effet, la notation pour spécifier une instance utilise le style souligné.

Dans la figure ci-dessous, deux parties sont également introduites, correspondant à chaque demi-arbre.

Entre chaque partie correspondant à une roue avant et chaque partie correspondant à un demi-arbre, un connecteur représente l'association liée. Un connecteur relie deux parties. Il est typé par une association inter objets, comme une partie est typée par une classe.

S'il existe plusieurs connecteurs entre deux parties et que ceux-ci sont typés par la même association, il est possible de les distinguer en leur attribuant des noms de rôle à l'instar des noms de rôle des parties.



Les connecteurs peuvent également relier des parties entre elles au travers de ports. Un port est un point d'interaction. Il possède une interface qui constitue son type et définit l'ensemble des interactions possibles. Les interactions conduites par un port se font avec les autres ports qui lui sont liés par un connecteur.

Un port peut également être introduit au niveau du classificateur. Un tel port a alors pour objectif de servir de passerelle entre les parties internes du classificateur et les objets externes à celui-ci.

Du point de vue de l'encapsulation, un tel port est généralement public. Il est alors connu en dehors du classificateur.

Remarque : Un port défini au niveau du classificateur peut aussi être défini comme privé. Il est alors réservé à une communication interne entre le classificateur et ses parties. Il ne fait pas office de passerelle entre l'intérieur et l'extérieur du classificateur.

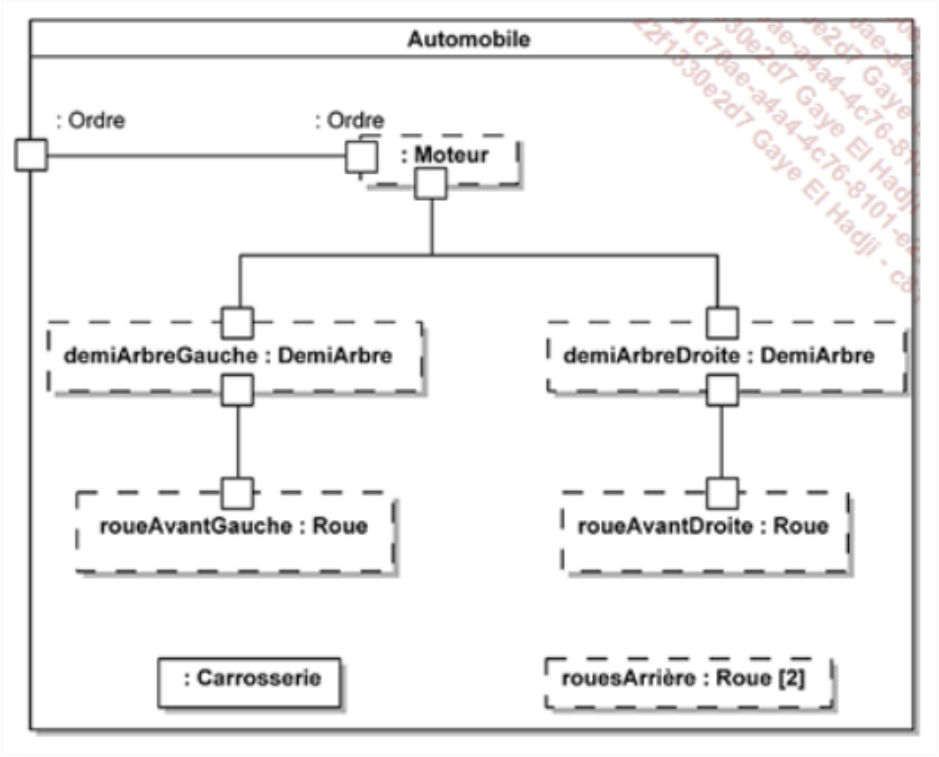
Une partie peut posséder plusieurs ports, chacun possédant sa propre interface. Plusieurs ports d'une même partie peuvent être typés avec la même interface. Il est alors possible de les distinguer en leur affectant des noms de rôle différents, à l'instar de ce qui se fait pour les parties et les connecteurs.

La figure ci-dessous illustre la même décomposition en parties de l'objet Automobile que dans le dernier exemple. Des connecteurs ont été ajoutés entre le moteur et les demi-arbres. Chaque connecteur entre les parties est relié au travers d'un port qui se présente sous la forme d'un carré blanc. Un port a également été ajouté au niveau du classificateur. Il est typé par l'interface Ordre. Il est connecté à un port du moteur également typé par cette interface.

Au niveau des interactions, cette figure illustre que :

- La classe Automobile peut interagir avec l'extérieur pour recevoir des ordres destinés à son moteur et qui lui sont transmis.
- Le moteur communique avec les demi-arbres (transmission du mouvement).
- Chaque demi-arbre communique avec les roues (transmission du mouvement).

Les connecteurs ne sont pas typés pour des raisons de simplification. De même, l'interface des ports des demi-arbres, des roues et du port du moteur connecté aux demi-arbres n'a pas été indiquée. Il pourrait s'agir de l'interface Transmission.

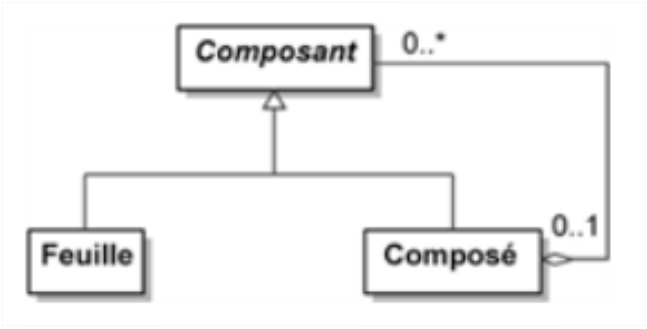


Une collaboration décrit un ensemble d'objets qui interagissent entre eux afin de réaliser une fonctionnalité d'un système. Chaque objet participe à cette fonctionnalité en effectuant une fonction précise.

Au sein d'une collaboration, les objets sont décrits comme dans un classificateur, avec les mêmes éléments : parties, connecteurs, ports...

Les collaborations peuvent être utilisées pour décrire les patterns de conception (design patterns) qui sont employés en programmation par objets.

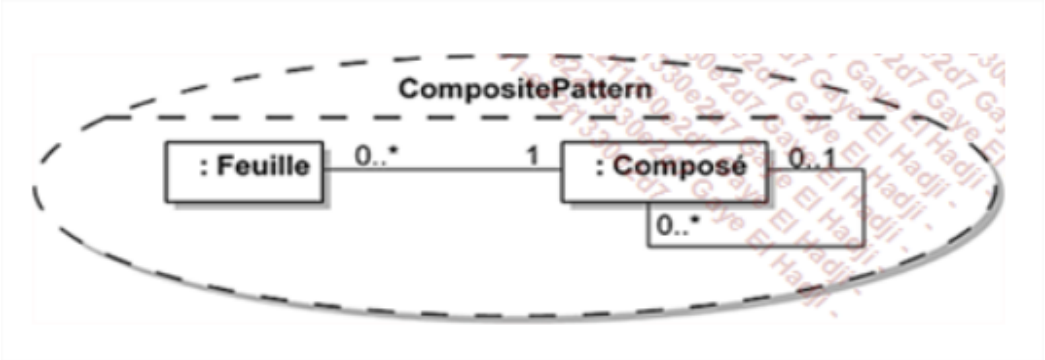
La figure ci-dessous montre le diagramme de classes de l'un de ces patterns, à savoir le pattern Composite.



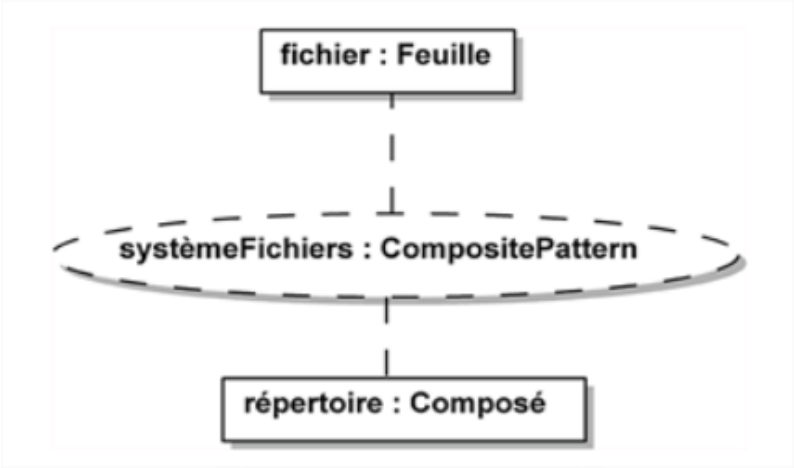
Le but du pattern Composite est d'offrir un cadre de conception d'une composition d'objets dont la profondeur est variable. Un composant est soit une feuille soit un objet composé, à son tour, de feuilles et d'autres composés. L'un des exemples qui illustrent le mieux ce pattern est celui du système de fichiers du disque dur d'un ordinateur personnel. Il est composé de fichiers et de répertoires. Chaque répertoire peut, à son tour, contenir des fichiers ou d'autres répertoires.

Une collaboration décrivant le pattern Composite est illustrée à la figure ci-dessous. Celle-ci présente les deux parties de la collaboration, à savoir celle qui a pour classe Feuille et celle qui a pour classe Composé. Toute feuille est nécessairement contenue dans un et un seul composé, c'est-à-dire qu'il existe au moins un composé. Un composé peut être contenu dans un autre composé ou non (cas du composé racine).

Remarque : Une collaboration ne remplace pas un diagramme de classes mais le complète. C'est ce que nous avons dit au début : le diagramme de structure composite n'a pas vocation à remplacer le diagramme de classes mais à le compléter.



Le diagramme de structure composite offre la possibilité de décrire une application d'une collaboration en fixant les rôles des parties, ce que nous avons fait à la figure ci-dessous en prenant notre exemple de système de fichiers comme application de la collaboration du pattern Composite.



12. Diagramme de déploiement

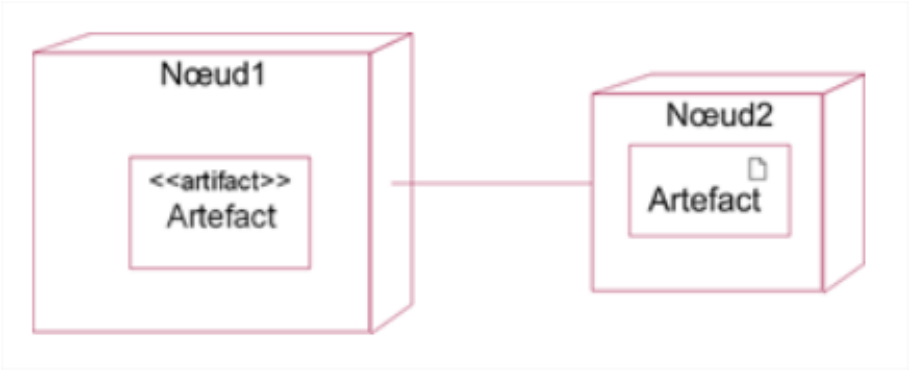
Le diagramme de déploiement décrit l'architecture physique du système. Celui-ci est composé de nœuds. Un nœud est une unité matérielle capable de recevoir et d'exécuter des éléments logiciels. La plupart des nœuds sont des ordinateurs. Les liaisons physiques entre nœuds peuvent également être décrites dans le diagramme de déploiement. Elles correspondent aux branches du réseau.

Les nœuds contiennent des éléments logiciels sous leur forme physique nommée artefact. Un fichier exécutable, une bibliothèque partagée ou un script sont des exemples de formes physiques des éléments logiciels.

Les composants qui constituent l'architecture logicielle du système sont représentés dans le diagramme de déploiement par un artefact qui est souvent un exécutable ou une bibliothèque partagée.

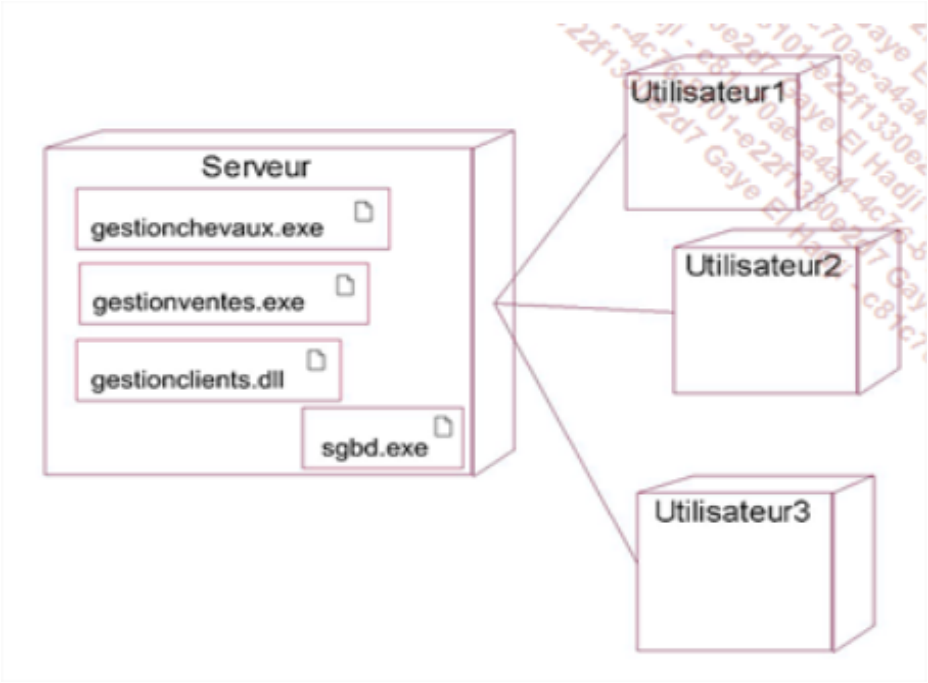
La représentation graphique des nœuds, de leurs liens et des artefacts qu'ils contiennent est illustrée à la figure 10.7. Le stéréotype « artifact » sert à préciser que l'élément est un artefact. Il peut également être représenté par l'icône d'un document.

Le mot "artifact" est le nom anglais pour "artefact".



Exemple : La figure ci-dessous montre l'architecture matérielle du système d'information d'un élevage de chevaux. Cette architecture est basée sur un serveur et trois postes clients. Ces derniers sont connectés au serveur par des liens directs. Le serveur contient plusieurs artefacts :

- Un exécutable (.exe), forme physique du composant de gestion de la base de données ;
- Un deuxième exécutable chargé de la gestion des chevaux ;
- Un troisième exécutable chargé de la gestion des ventes ;
- Une bibliothèque partagée (.dll) de gestion des machines des différents utilisateurs.



Il est également possible de représenter l'architecture d'un système particulier. UML offre la possibilité de décrire des instances de nœuds ou d'artefacts.

13. Diagramme de profils

En UML, un diagramme de profil est une vue statique qui permet de décrire un mécanisme d'extension léger par rapport à UML pour répondre à un domaine particulier par exemple Java ou JEE ou .Net. Par exemple en Java l'héritage multiple n'existe pas donc dans notre diagramme de profil nous allons obliger cette restriction.

14. Diagramme global d'interaction

Le Diagramme global d'interaction ou diagramme d'interactivité est un diagramme UML version 2.0 utilisé pour rendre compte de l'organisation spatiale des participants à l'interaction.
Les diagrammes globaux d'interaction définissent des interactions par une variante des diagrammes d'activité, d'une manière qui permet une vue d'ensemble de flux de contrôle.
Ils se concentrent sur la vue d'ensemble de flux de contrôle où les nœuds sont des interactions ou InteractionUses.
Les lignes de vie et les messages n'apparaissent pas à ce niveau de vue d'ensemble.