

Ressources Informatiques

UML 2

**Initiation,
exemples
et exercices corrigés**

**Seconde
Edition**

Laurent DEBRAUWER
et Fien VAN DER HEYDE

INFORMATIQUE TECHNIQUE



UML 2

Initiation, exemples et exercices corrigés [2ième édition]

Fien VAN DER HEYDE - Laurent DEBRAUWER



Résumé

Ce livre sur UML 2 s'adresse tout autant aux **étudiants** qu'aux **développeurs** pratiquant la **modélisation de systèmes et de processus**. Vous découvrirez, étape par étape, les éléments de modélisation à partir d'**exemples pédagogiques** issus... du monde des chevaux. Après une introduction à l'**approche par objets**, cet ouvrage introduit les différents diagrammes d'UML 2 depuis la **description des exigences** par les cas d'utilisation jusqu'au **diagramme des composants** en passant par les **diagrammes d'interaction, de classes, d'états transitions et d'activités**. Vous apprendrez comment les **diagrammes d'interaction** peuvent être utilisés pour découvrir les objets composant le système.

L'auteur

Fien Van der Heyde, de formation supérieure financière et informatique, exerce le métier de responsable informatique d'une grande banque au Luxembourg. La modélisation des processus tient une place importante dans ses activités professionnelles mais elle s'intéresse aussi beaucoup... au monde du cheval.

Laurent Debrauwer, docteur en informatique de l'Université de Lille 1, auteur de logiciels dans le domaine de la linguistique, exerce le métier de consultant indépendant en tant que spécialiste de l'approche par objets.

Ce livre numérique a été conçu et est diffusé dans le respect des droits d'auteur. Toutes les marques citées ont été déposées par leur éditeur respectif. La loi du 11 Mars 1957 n'autorisant aux termes des alinéas 2 et 3 de l'article 41, d'une part, que les "copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective", et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, "toute représentation ou reproduction intégrale, ou partielle, faite sans le consentement de l'auteur ou de ses ayants droit ou ayant cause, est illicite" (alinéa 1er de l'article 40). Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles 425 et suivants du Code Pénal.

Copyright Editions ENI

Pourquoi ce livre ?

UML (*Unified Modeling Language* ou langage unifié de modélisation) est un langage graphique destiné à la modélisation de systèmes et de processus. UML est un langage basé sur l'approche par objets, celle qui a d'abord conduit à la création des langages de programmation comme Java, C++ ou Smalltalk.

UML est unifié car il provient de plusieurs notations qui l'ont précédé. Aujourd'hui, UML est promu par l'OMG (*Object Management Group*), un consortium de plus de 800 sociétés et universités actives dans le domaine des technologies de l'objet.

Notre idée est qu'UML deviendra un langage de modélisation très répandu, notamment grâce à sa richesse sémantique qui le rend abstrait de nombreux aspects techniques. Le premier objectif de cet ouvrage est la vulgarisation de UML. Ce livre est donc destiné au plus grand nombre, informaticien, chef de projet, manager ou toute personne souhaitant disposer, en plusieurs schémas, d'une vue d'ensemble d'un système.

Afin d'atteindre cet objectif, nous avons adopté la démarche suivante :

- expliquer chaque concept de la façon la plus simple et la plus complète possible ;
- introduire un grand nombre d'exemples afin d'illustrer au mieux notre propos ;
- ne pas fournir d'explications basées sur du code C++, Java ou issu d'un autre langage de programmation ;
- éviter les exemples classiques et choisir un domaine riche et rarement abordé, celui des chevaux ;
- proposer des exercices dont vous trouverez les solutions à la fin de l'ouvrage.

UML est sémantiquement riche, il est donc assez difficile de retenir tous ses concepts. Nous espérons que cet ouvrage vous sera utile pour les apprécier et les mémoriser. Lorsque vous serez amené à modéliser en UML, nous espérons également que le présent ouvrage vous servira de référence.

Ce livre a pour titre *UML 2 : Initiation, exemples et exercices corrigés*. Vous découvrirez au chapitre 2 l'historique d'UML. Vous y apprendrez que la version 2 est la version actuelle.

Le monde équin

Tous les exemples de notre ouvrage appartiennent au monde équin. Nous ne vous cacherons pas plus longtemps que Fien, coauteur, est passionnée par le monde du cheval et qu'elle possède une jument à tête busquée dénommée Jorphée photographiée à la figure 1.1.

Imaginez-vous, le temps de la lecture de cet ouvrage, à la tête d'un élevage de chevaux. La scène pourrait se dérouler dans le Kentucky. Votre ranch serait constitué d'un troupeau de quarter horse. Vous préférez le côté chic de Deauville. Dans ce cas, vous seriez à la tête d'un haras en Normandie constitué de pur-sang anglais destinés à l'élevage de yearlings. Quel que soit votre choix, vous devrez assurer un suivi rigoureux et gérer une très grande quantité de données. Ces dernières entretiennent entre elles des liens spécifiques et ont des cycles propres.

Vous pourrez, par exemple, être amené à vous poser ces questions quotidiennement :

- La jument Jorphée : N'est-ce pas aujourd'hui qu'elle doit pouliner ?
- Le cheval Espiègle : Quelle ration d'avoine dois-je lui donner ?
- Le cheval Quincy : Quand dois-je le vacciner ?



Figure 1.1 - Jorphée du GISORS

Le contenu de l'ouvrage

L'ouvrage est organisé en dix chapitres dont nous donnons ci-après un bref descriptif.

Chapitre 1 - Introduction

Il s'agit de la présente introduction.

Chapitre 2 - À propos d'UML

Ce chapitre est consacré d'une part à la genèse d'UML et d'autre part au RUP (*Rational Unified Process* ou processus uniifié de Rational) et à MDA (*Model Driven Architecture* ou architecture guidée par les modèles).

RUP est un processus de développement et d'évolution de logiciels. L'architecture MDA est destinée à la réalisation de logiciels en faisant abstraction de la plate-forme et du langage de programmation.

Chapitre 3 - Les concepts de l'approche par objets

Le chapitre 3 est dédié à la découverte des 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.

Chapitre 4 - La modélisation des exigences

Le chapitre 4 a pour objectif de vous faire découvrir les cas d'utilisation qui sont employés pour décrire les exigences fonctionnelles attendues lors de la rédaction du cahier des charges d'un système ou encore les fonctionnalités d'un système existant.

Chapitre 5 - La modélisation de la dynamique

Le chapitre 5 explique comment UML représente les interactions entre les objets. La description des interactions est également utilisée pour découvrir les objets composant un système. Cette découverte est fondée sur les interactions intervenant dans les cas d'utilisation du système.

Chapitre 6 - La modélisation des objets

Le chapitre 6 est essentiel. Il est consacré à la modélisation statique des objets, c'est-à-dire sans description des interactions ou du cycle de vie des objets. Les méthodes sont introduites d'un point de vue statique, sans description de leur enchaînement.

Ce chapitre inclut l'étude du diagramme des classes. Ce diagramme contient les attributs, les méthodes et les associations des objets. Il est central lors d'une modélisation par objets d'un système. De tous les diagrammes UML, il est le seul à être obligatoire lors d'une telle modélisation.

Chapitre 7 - La structuration des éléments de modélisation

Le chapitre 7 est consacré aux paquetages. UML 2 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.

Chapitre 8 - La modélisation du cycle de vie des objets

Le chapitre 8 étudie le cycle de vie des objets. Le cycle de vie d'un objet représente les différentes étapes ou états que celui-ci va suivre pour concourir, au sein du système, à la réalisation d'un objectif. Un état correspond à un moment d'activité ou d'inactivité (attente) de l'objet.

Chapitre 9 - La modélisation des activités

Le chapitre 9 est dédié au diagramme d'activités. Celui-ci s'appuie sur le diagramme d'états-transitions étudié au chapitre 8. Il s'agit d'une forme spécifique du diagramme d'états-transitions dans lequel tous les états sont associés à une activité et toutes les transitions sont automatiques. Les transitions sont appelées enchaînements dans ce diagramme.

Chapitre 10 - La modélisation de l'architecture du système

Le chapitre 10 est consacré à la modélisation de l'architecture du système. Cette modélisation se décline sous deux aspects :

- la modélisation de l'architecture logicielle et sa structuration en com- posants ;
- la modélisation de l'architecture matérielle et la répartition physique des logiciels.

Annexe 1 - L'architecture MDA : l'outil DB-MAIN

L'annexe 1 présente l'outil DB-MAIN dans le cadre de l'architecture MDA appliquée à la réalisation de schémas de bases de données relationnelles.

Annexe 2 - Correction des exercices

L'annexe 2 détaille une correction possible des exercices introduits à la fin de certains chapitres.

Annexe 3 - Glossaire

L'annexe 3 est un glossaire des différents termes employés dans le présent ouvrage.

Annexe 4 - Lexique français-anglais/anglais-français

L'annexe 4 présente un lexique français-anglais et anglais-français des différents termes employés dans le présent ouvrage.

Annexe 5 - Notation graphique

L'annexe 5 est un résumé de la notation graphique des principaux éléments d'UML.

Annexe 6 - Bibliographie

L'annexe 6 est une bibliographie des principaux ouvrages de référence de la notation UML.

Introduction

Ce chapitre est consacré, d'une part, à la genèse d'UML et d'autre part à deux approches connexes à UML :

- RUP (*Rational Unified Process* ou processus unifié de Rational) processus de développement et d'évolution de logiciels ;
- l'architecture MDA (*Model Driven Architecture* ou architecture guidée par les modèles) destinée à la réalisation de systèmes en faisant abstraction de la plate-forme physique et de ses aspects technologiques.

La genèse d'UML : Unified Modeling Language

UML est basé sur l'approche par objets. Cette dernière voit 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 connaît de nombreux successeurs : Smalltalk, C++, Java ou plus récemment C#.

Dans un langage de programmation, la description des objets est réalisée de façon formelle avec une syntaxe rigoureuse. Cette syntaxe n'est pas lisible par des non-programmeurs et reste difficile à déchiffrer pour des programmeurs. À la différence des machines, les humains préfèrent utiliser les langages graphiques pour représenter des abstractions. Ils maîtrisent ce type de langage plus facilement et obtiennent une vue d'ensemble d'un système en un temps beaucoup plus court.

Dans les années 1980 et au début des années 1990, les notations graphiques se multiplient, chacun utilisant bien souvent sa propre notation. En 1994, James Rumbaugh et Grady Booch décident de se regrouper pour unifier leurs notations. Celles-ci proviennent de leurs méthodes : OMT pour James Rumbaugh et méthode Booch pour Grady Booch. En 1995, Yvar Jacobson décide de rejoindre l'équipe des "trois amis". Cette équipe travaille alors au sein de Rational Software.

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

Depuis cette époque, cette Task Force a mis à jour UML plusieurs fois. En mars 2003, la version 1.5 voit le jour. Cette dernière offre la possibilité de décrire des actions grâce à une extension d'UML appelée *Action Semantics* ou sémantique des actions dont l'étude dépasse le cadre de cet ouvrage.

La version d'UML dont traite cet ouvrage est la version 2.1 qui est désormais sous sa forme définitive. 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.

À l'aide d'un diagramme d'activité (chapitre La modélisation des activités), la figure 1 illustre l'évolution d'UML et en retrace la genèse ainsi que les principales versions.

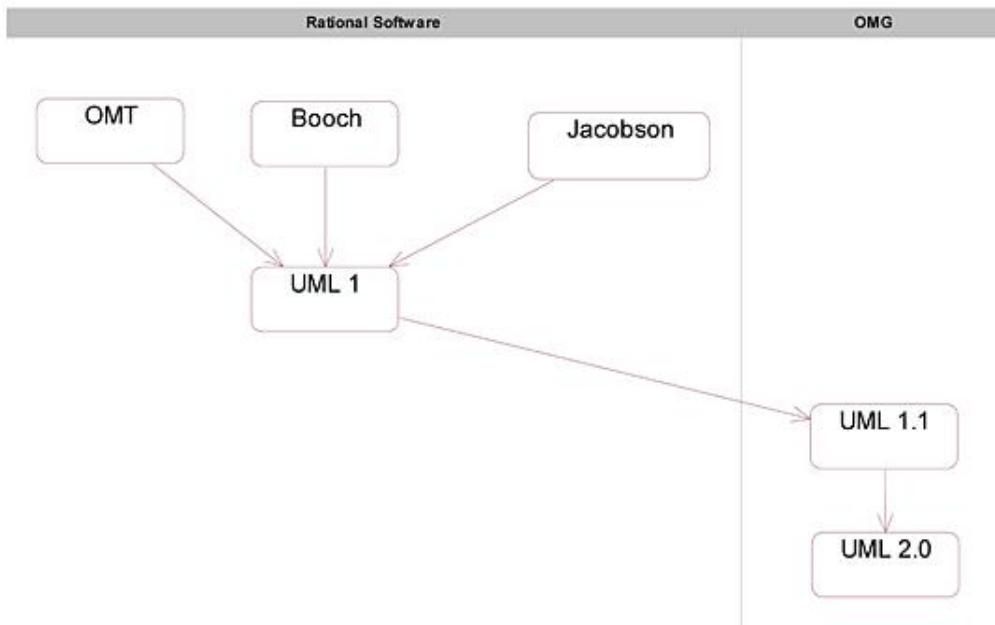


Figure 2.1 - Genèse et principales versions d'UML

Rappelons qu'UML est une notation destinée à la modélisation par objets de systèmes et de processus. UML ne contient pas un guide méthodologique mais constitue un support de modélisation.

RUP : Rational Unified Process

RUP est un processus de réalisation ou d'évolution de logiciels entièrement basé sur UML, d'où l'intérêt de le présenter dans cet ouvrage. RUP est constitué d'un ensemble de directives permettant de produire un logiciel à partir du cahier des charges (exigences). Chaque directive définit *qui fait quoi et à quel moment*. Un processus permet donc de structurer les différentes étapes d'un projet informatique.

RUP est un produit commercial fourni sous la forme d'un site web réservé aux clients qui en ont acquis la licence auprès de Rational Software (<http://www.rational.com>). Les principes de RUP sont issus du Processus Unifié ; les trois auteurs de ce dernier sont les mêmes que ceux d'UML. La différence entre RUP et le Processus Unifié est qu'il existe un grand nombre de modèles disponibles offrant un gain de temps dans RUP. D'autre part, RUP peut être étendu à l'aide de plugins disponibles sur le marché en vue de l'adapter à des besoins spécifiques.

L'utilisation d'UML n'impose pas celle de RUP. On peut employer un autre processus ou même aucun processus.

RUP est piloté par les cas d'utilisation que nous étudierons au chapitre La modélisation des exigences. Ceux-ci sont utilisés pour décrire les exigences du projet. Ils sont décrits à l'aide d'une représentation spécifique à RUP plus riche que celle contenue dans UML.

RUP est incrémental. Un projet est divisé en une suite de sous-projets. Chaque sous-projet est une brique ajoutée au sous-projet précédent qui doit donc avoir été préalablement réalisé. Quand le dernier sous-projet est réalisé, le projet dans son intégralité est donc achevé.

RUP est itératif. Chaque sous-projet est réalisé avec les mêmes activités. À l'issue de chaque sous-projet, une livraison partielle est évaluée.

Les créateurs de RUP proposent cette approche incrémentale et itérative pour éviter de traiter un projet important dans sa globalité avec une livraison intervenant longtemps après la rédaction du cahier des charges. En effet, dans un tel cas, il est facile d'imaginer que les besoins du client auront évolué et qu'il n'aura guère de souvenirs de ce qu'il avait demandé dans le cahier des charges. Un conflit peut alors se produire alors qu'une approche incrémentale et itérative aurait permis de l'éviter.

Le cycle de développement est divisé en quatre phases :

- La phase *d'inception* consiste à évaluer le projet. On décide de poursuivre ou non le projet en fonction des impératifs financiers. Les principaux cas d'utilisation et une première ébauche d'architecture sont déterminés.
- L'élaboration a pour objectif de construire l'architecture du système. À l'issue de l'élaboration, les exigences du projet et l'architecture sont définitivement connues.
- La construction correspond au développement logiciel de l'architecture déterminée lors de la phase d'élaboration.
- La transition comprend le déploiement du logiciel chez le client et la formation des utilisateurs.

Dans RUP, chaque phase est détaillée par un ensemble d'activités. Une activité est un ensemble d'actions décrit par un diagramme d'activités comme nous les examinerons au chapitre La modélisation des activités. Avec RUP, un dictionnaire très riche constitué de modèles d'activités et de cas d'utilisation adaptés à des secteurs d'activité spécifiques est également fourni.

Les principales activités de RUP sont les suivantes :

- modélisation des processus *métier* ;
- gestion des exigences ;
- analyse et conception ;
- implantation et test ;
- déploiement.

Dans la phase *d'inception*, les activités les plus couramment utilisées sont la modélisation des processus *métier* et la gestion des exigences.

Lors de l'élaboration, les activités les plus fréquemment employées sont la gestion des exigences ainsi que l'analyse et la conception.

La construction comprend principalement l'analyse et la conception ainsi que l'implantation et le test.

La phase de transition fait surtout appel à l'activité de déploiement.

En conclusion, RUP est une méthode itérative et agile de développement. Elle se distingue ainsi des approches classiques comme le cycle en cascade qui va séquentiellement de l'écriture des besoins à la livraison.

MDA : Model Driven Architecture

MDA est une nouvelle proposition de l'OMG dont l'objectif est la conception de systèmes basée sur la seule modélisation du domaine, en faisant abstraction des aspects technologiques. À partir de cette modélisation, MDA propose d'obtenir par transformation les éléments techniques capables de fonctionner au sein d'une plateforme logicielle comme Java ou .NET.

Dans MDA, le modèle des objets du domaine s'appelle PIM, c'est-à-dire *Platform Independent Model* ou modèle indépendant de la plateforme. Le PIM est constitué d'un ensemble d'éléments dont la conception doit se faire indépendamment de tout langage de programmation ou de technologie. Ce modèle est ensuite transformé manuellement ou automatiquement en un modèle spécifique à une plateforme et à un langage de programmation. Un tel modèle spécifique s'appelle PSM, c'est-à-dire *Platform Specific Model* ou modèle spécifique à la plateforme.

Le lien avec UML réside au niveau du PIM. En effet, UML est un très bon candidat comme langage à ce niveau. UML possède l'avantage de décrire finement des objets tout en restant indépendant des technologies. Au niveau des traitements, l'extension Action Semantics d'UML devrait lui permettre de répondre à tous les besoins de description.

Introduction

Ce chapitre 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 aborderons le concept d'objet, puis nous verrons, par abstraction, comment le modéliser 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.

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.

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.

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 3.1.

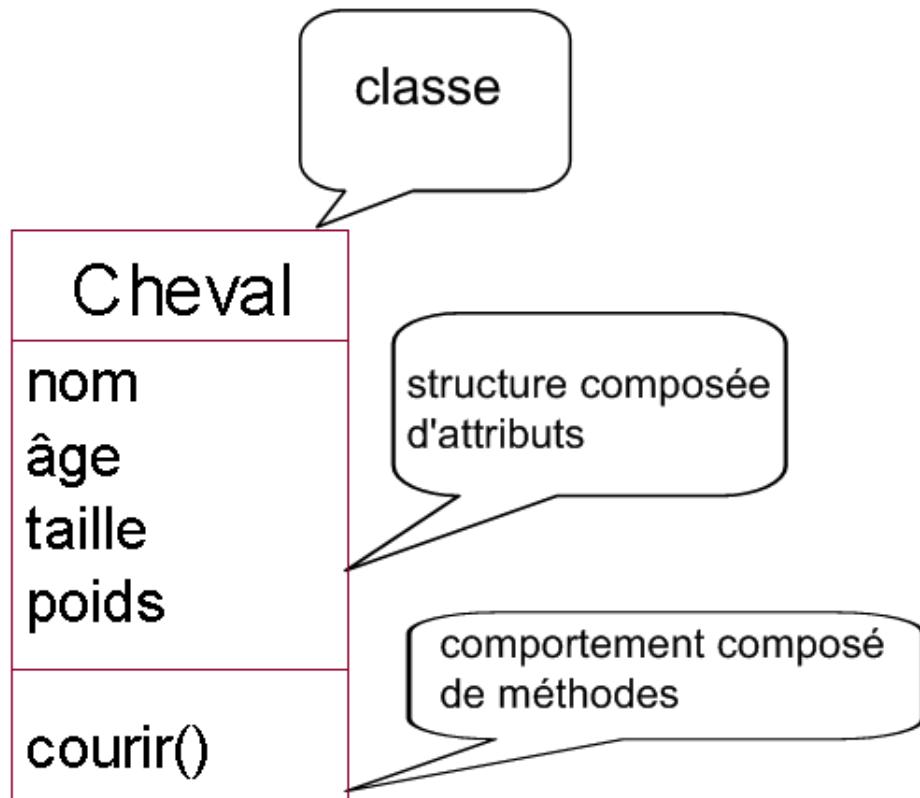


Figure 3.1 - La classe *Cheval*

Le cheval *Jorphée* est une instance de la classe *Cheval* dont les attributs et leurs valeurs sont illustrés à la figure 3.2.

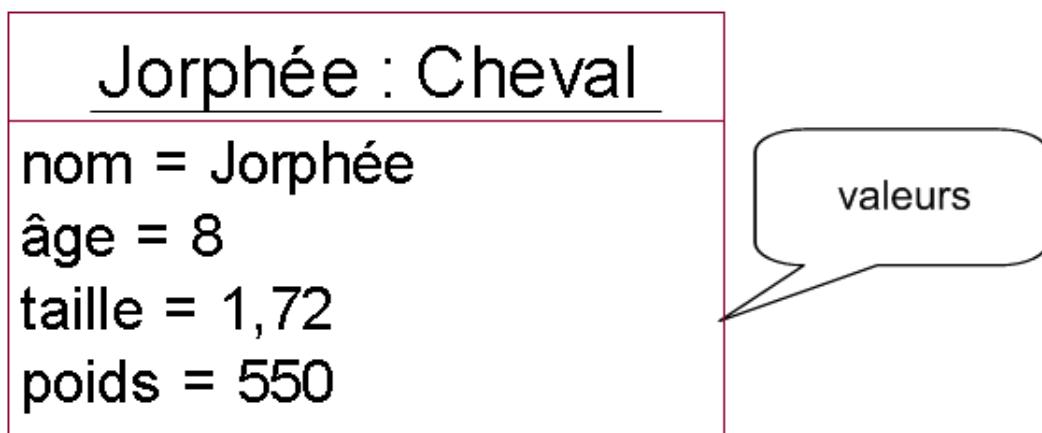


Figure 3.2 - Le cheval *Jorphée*

➤ Le nom d'une classe apparaît au singulier. Il est toujours 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.

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 publiques 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 va effectuer 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 3.3.

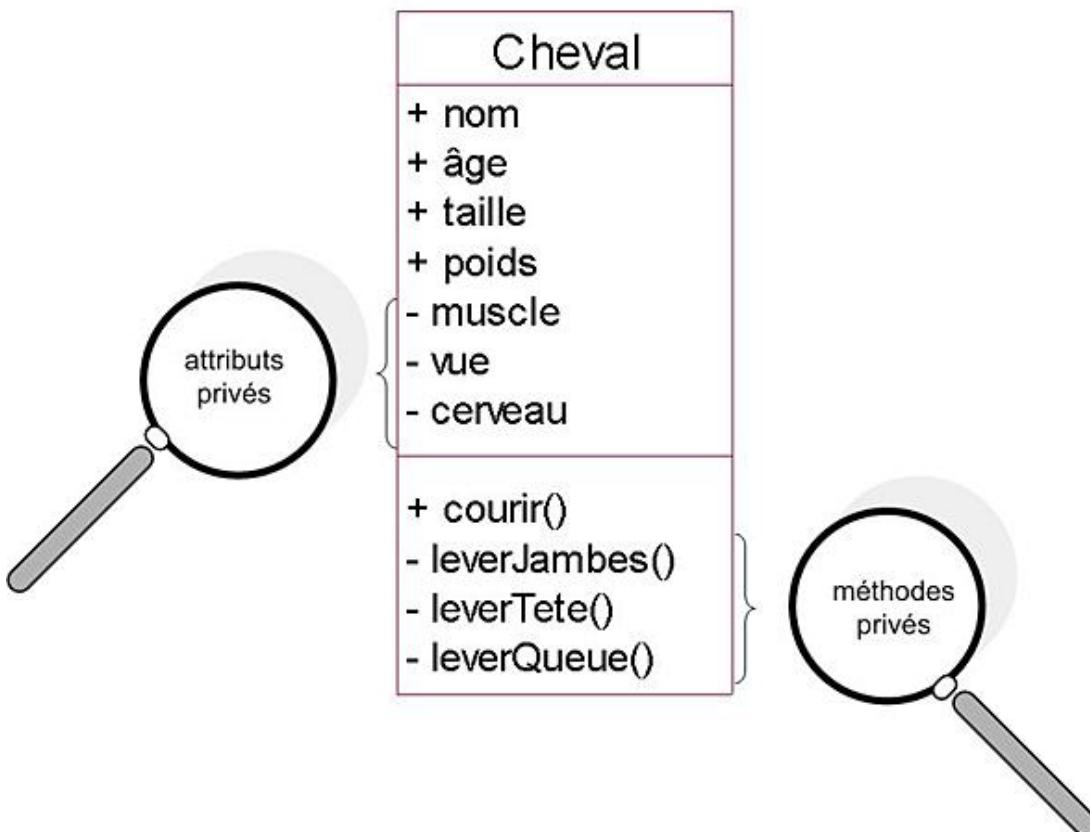


Figure 3.3 - La classe Cheval détaillée

-
- 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.
-

La spécialisation et la généralisation

Jusqu'à présent, chaque classe d'objets est introduite séparément des autres classes. Une classe peut également être 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 est une spécialisation d'une autre classe, cette dernière est une généralisation de la première. Elle en est sa surclasse.

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 3.4.

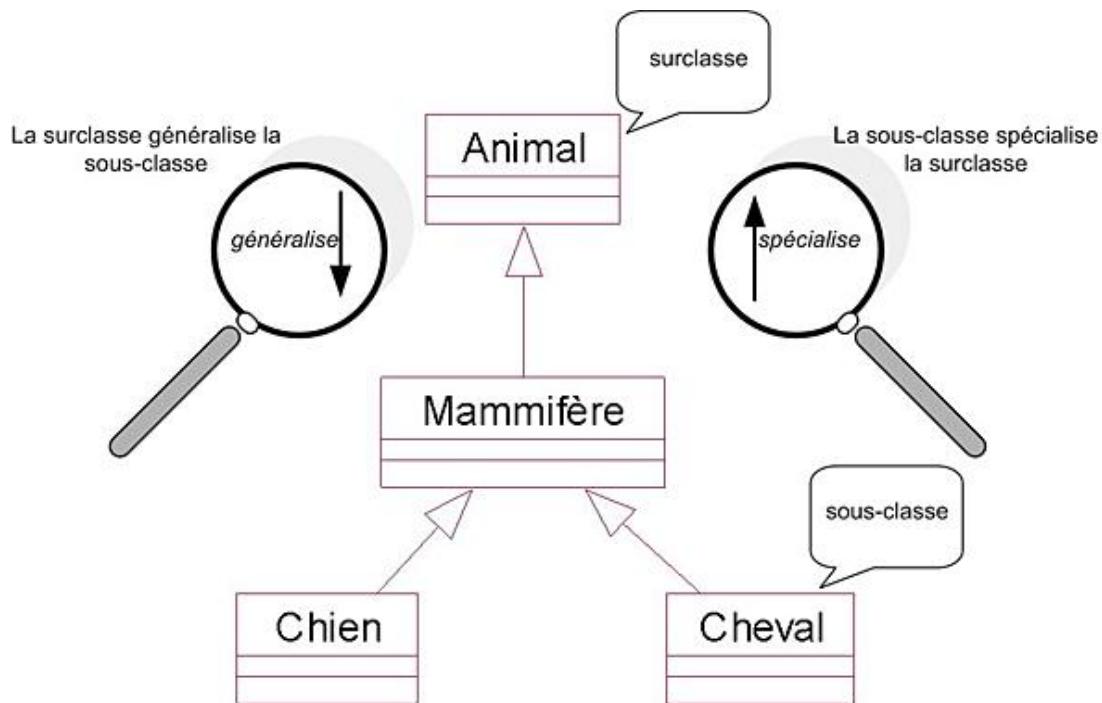


Figure 3.4 - Hiérarchie de classes

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 3.5.

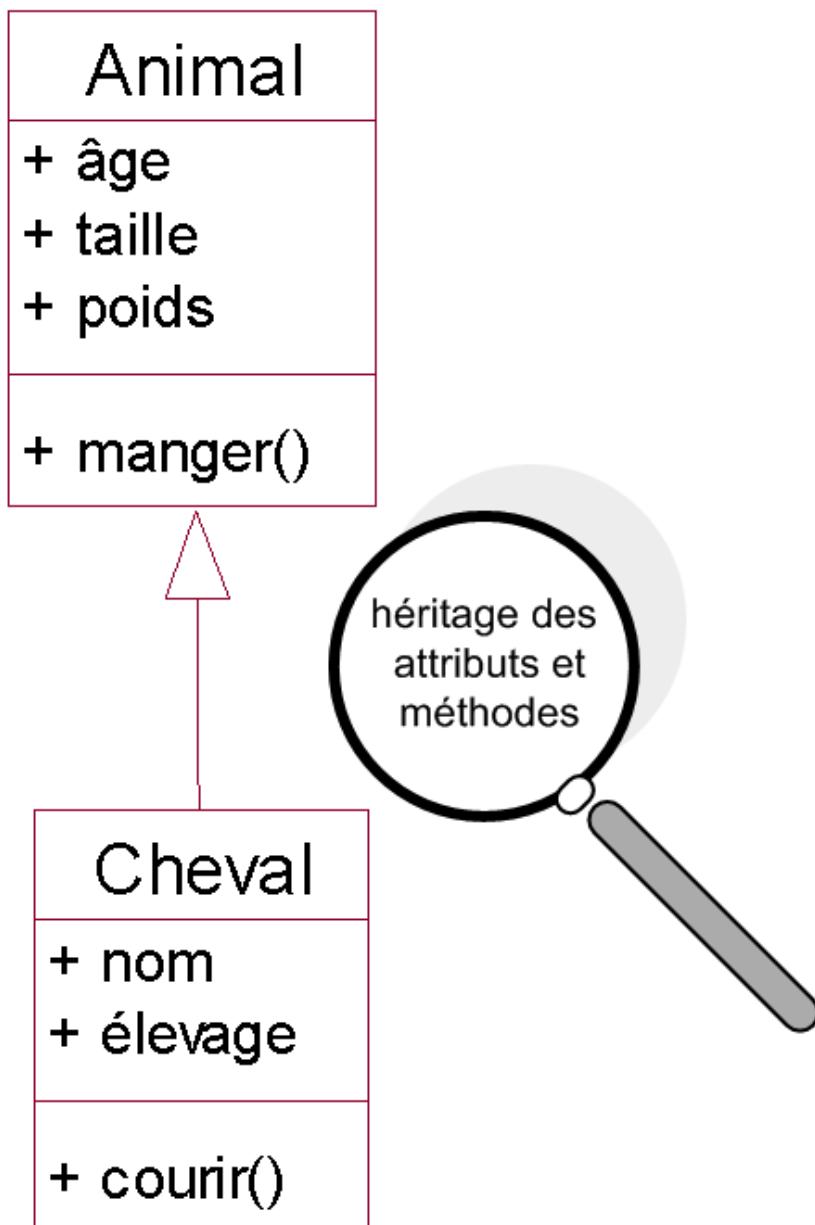


Figure 3.5 - Héritage

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.

Les classes abstraites et concrètes

L'examen de la hiérarchie présentée à la figure 3.4 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 3.6 reprend la hiérarchie en indiquant précisément les classes abstraites et les classes concrètes. En UML, le nom des classes abstraites apparaît en caractères italiques.

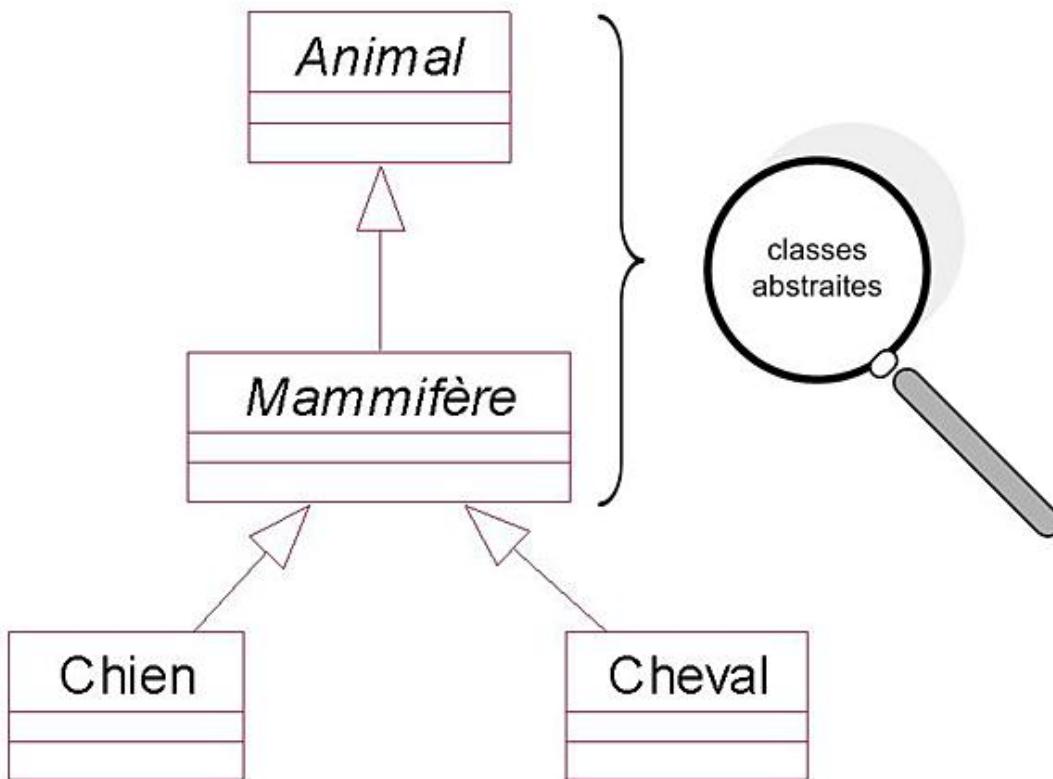


Figure 3.6 - Classes abstraites et classes concrètes

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 3.7. 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.

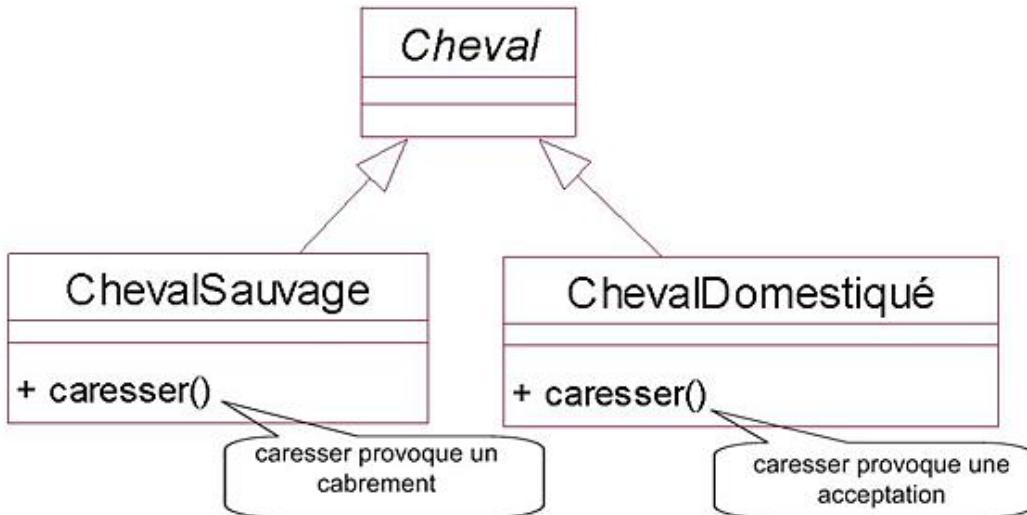


Figure 3.7 - Polymorphisme

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 3.8.

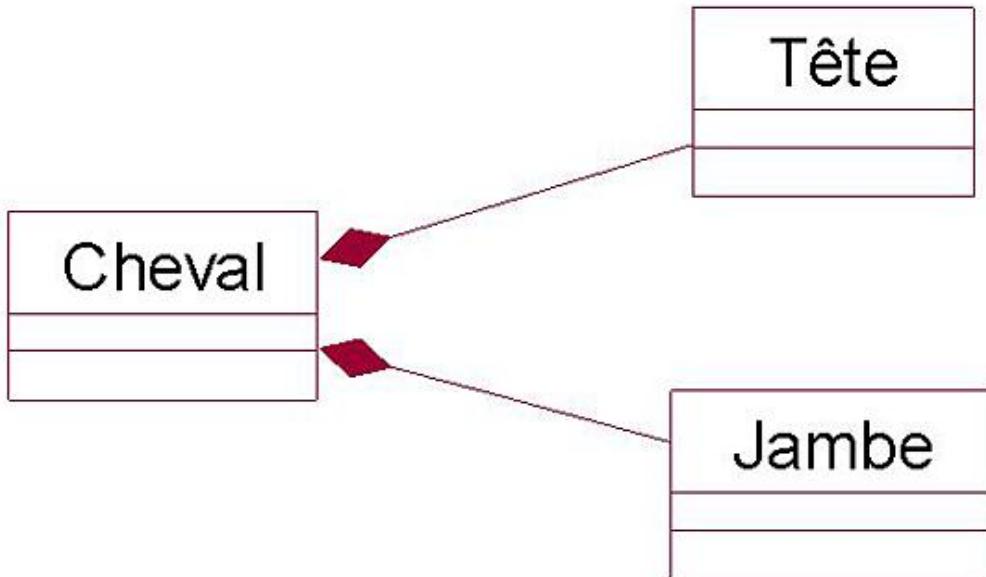


Figure 3.8 - Composition

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 3.9.

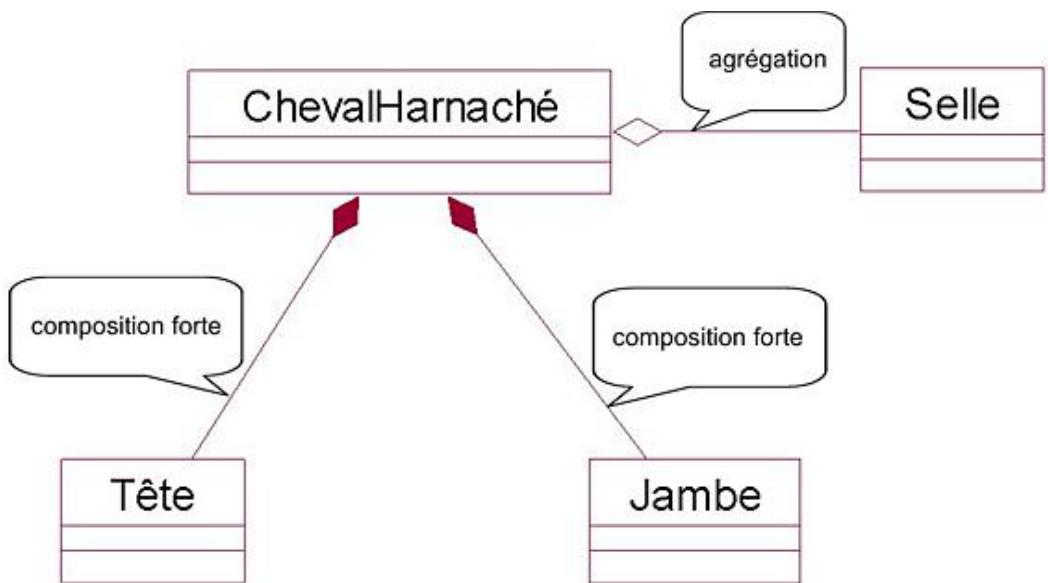


Figure 3.9 - Composition et agrégation

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 italiques. Cette représentation graphique inclut un stéréotype implicite, mais il est également possible de ne pas mettre le nom de la classe en italiques et de préciser explicitement le stéréotype «abstract».

Ce stéréotype explicite est illustré à la figure 3.10. Il peut être employé lors de l'écriture manuelle de diagrammes UML.

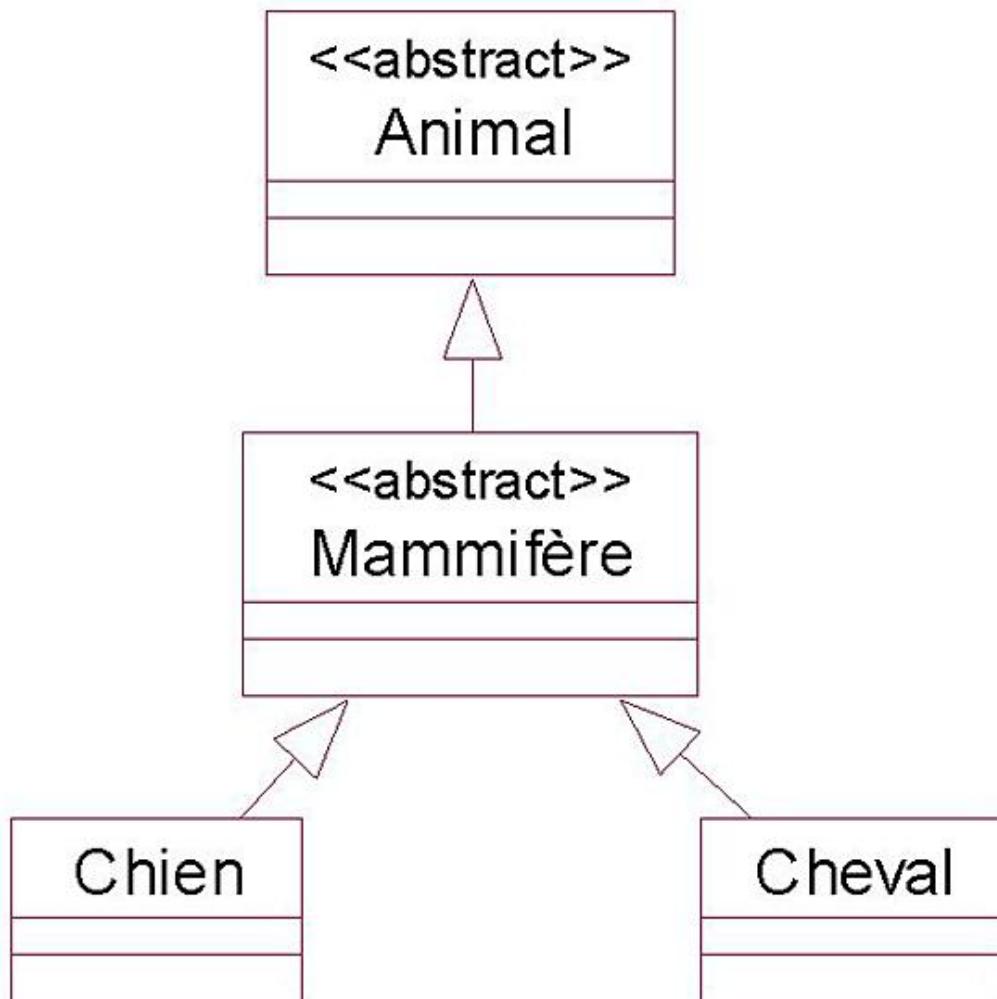


Figure 3.10 - Stéréotype explicite «abstract»

Ce schéma est équivalent au schéma de la figure 3.11 déjà introduit à la figure 3.6.

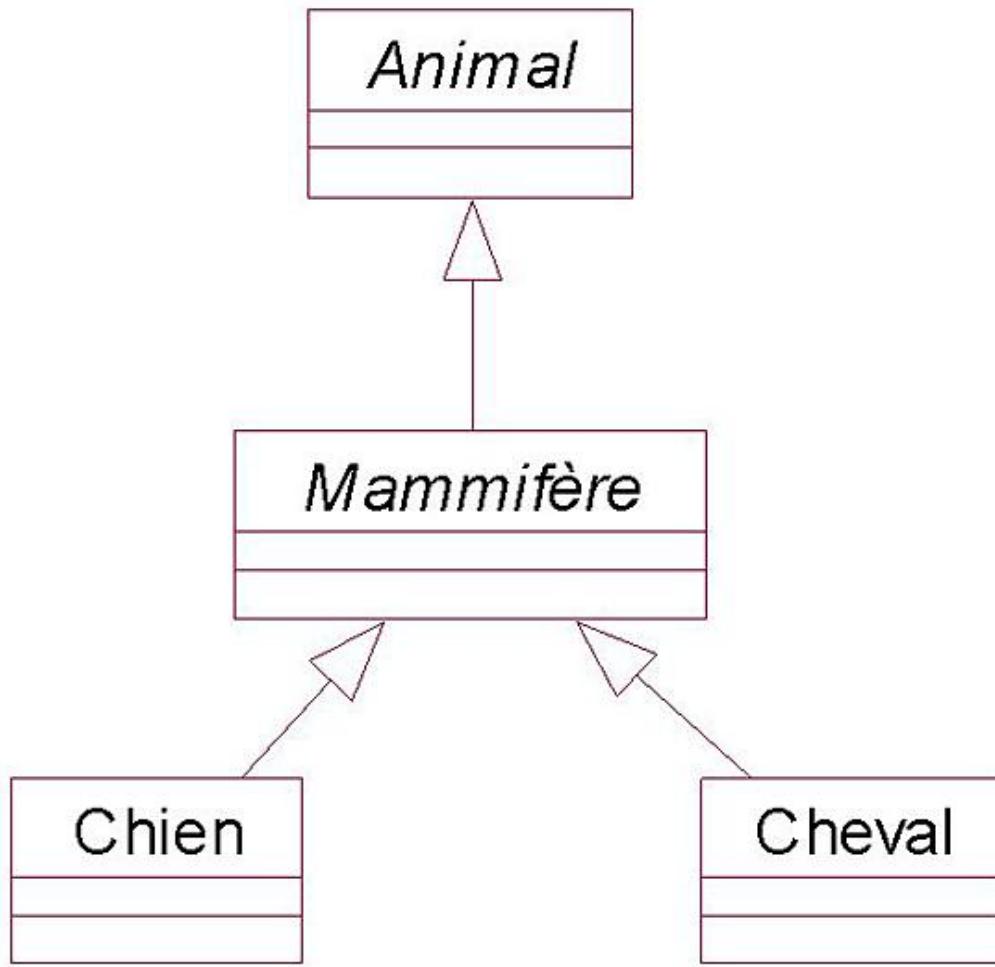


Figure 3.11 - Stéréotype implicite «abstract»

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.

Introduction

Ce chapitre a pour objectif de 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 (utilisateurs du système) ainsi que les relations 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 les clients et, par conséquent, une base pour l'élaboration au niveau fonctionnel du cahier des charges.

Cas d'utilisation

Les cas d'utilisation décrivent sous la forme d'actions et de réactions, le comportement du système étudié du point de vue des utilisateurs. 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 utilisateur et le système. Elle peut être formulée ainsi :

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

Un cas d'utilisation explicite la partie des exigences fonctionnelles du système concernant l'un des objectifs d'un utilisateur. Ce dernier est aussi appelé, de façon plus précise, cas d'utilisation avec objectif utilisateur.

Exemple :

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

Acteur

Un utilisateur externe du système peut jouer différents rôles vis-à-vis du système. Un couple (utilisateur, rôle) constitue un acteur spécifique désigné en UML uniquement par le nom du rôle.

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

Deux catégories d'acteurs doivent être distinguées :

- les acteurs primaires, pour lesquels l'objectif du cas d'utilisation est essentiel ;
- les acteurs secondaires qui interagissent avec le cas d'utilisation mais dont l'objectif n'est pas essentiel.

Exemple :

Reprendons 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 constituent un acteur secondaire.

Scénario

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.

Relation de communication

La relation qui lie un acteur à un cas d'utilisation s'appelle la relation de communication.

Cette relation supporte différents modèles de communication, par exemple :

- les services que le système doit fournir à chacun des acteurs du cas d'utilisation ;
- les informations du système qu'un acteur peut introduire, consulter ou modifier ;
- les changements intervenant dans l'environnement dont un acteur informe le système ;
- les changements intervenant au sein du système dont ce dernier informe un acteur.

Exemple :

Lorsque Fien a acquis Jorphée, elle a reçu des informations de l'élevage comme la proposition de prix, les papiers de la jument, son carnet de vaccination et a fourni des informations comme une contre-proposition de prix, une promesse d'achat.

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 relations de communication qui les relient.

Exemple :

Le cas d'utilisation de l'achat d'un cheval est représenté par la figure 4.1 :

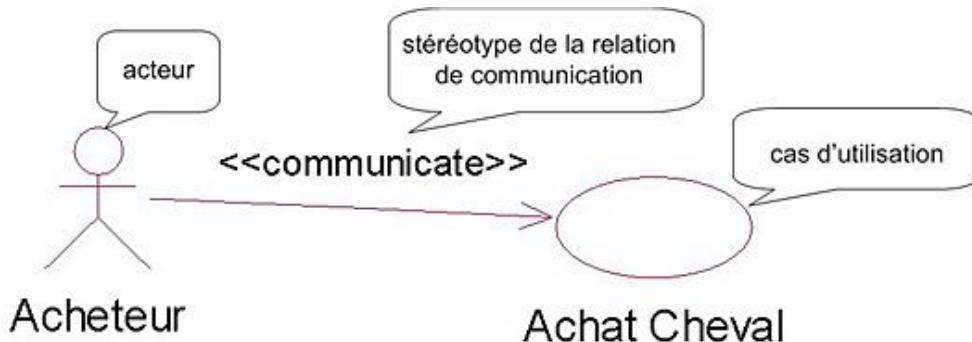


Figure 4.1 - Le cas d'utilisation d'achat d'un cheval

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, c'est-à-dire l'élevage de chevaux, est illustré à la figure 4.2.

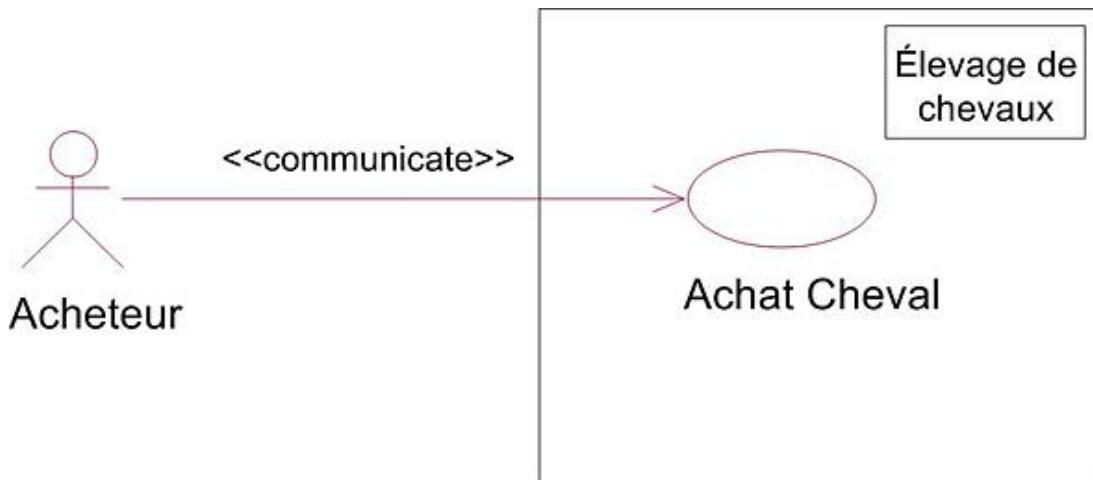


Figure 4.2 - Système d'un cas d'utilisation

Un acteur secondaire est représenté comme un acteur primaire. Souvent, le sens de la relation de communication entre un acteur secondaire et le système est inversé par rapport au sens de la relation entre un acteur primaire et le système. En effet, la communication est initiée par le système et non par l'acteur.

Exemple :

Dans l'exemple précédent, le changement de propriétaire du cheval est réalisé par les haras nationaux. Ces derniers constituent un acteur secondaire (voir figure 4.3).

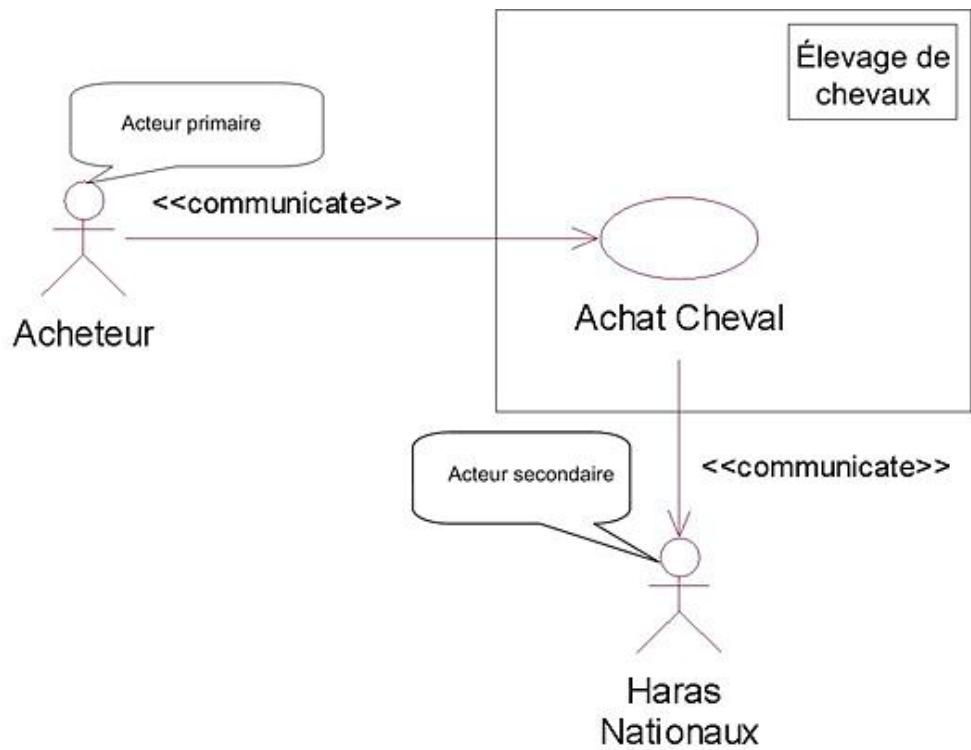


Figure 4.3 - Acteurs primaire et secondaire d'un cas d'utilisation

Les relations entre les cas d'utilisation

1. 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 va vérifier ses vaccinations. Par conséquent, le cas d'utilisation d'achat d'un étalon inclut cette vérification (voir figure 4.4).

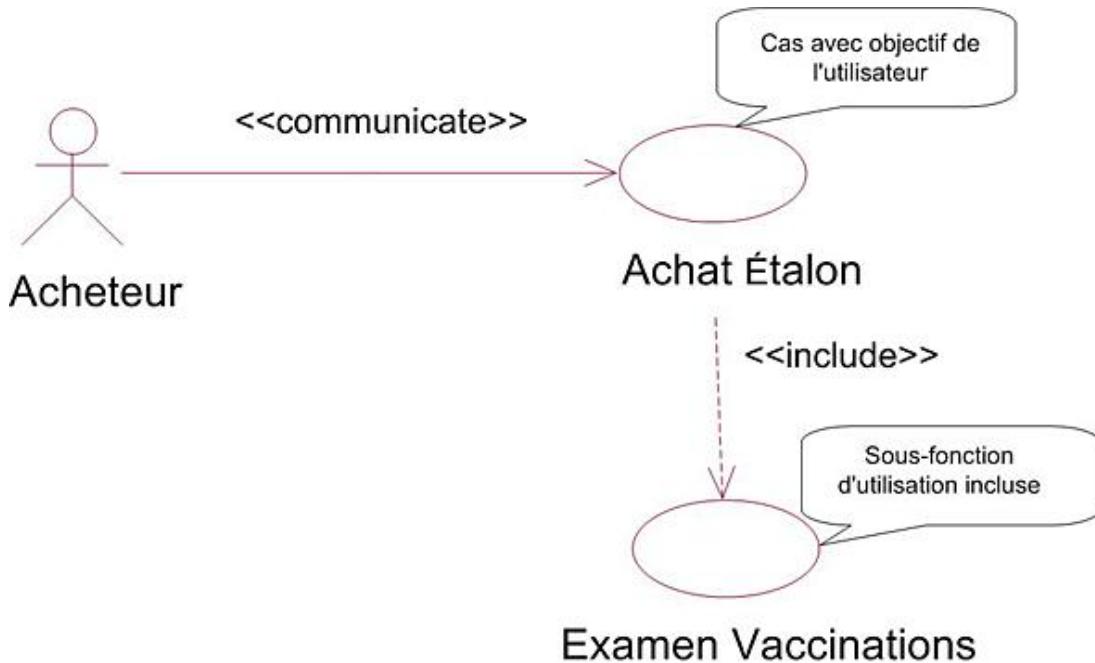


Figure 4.4 - Inclusion d'un cas d'utilisation

La mise en commun du cas d'utilisation d'examen des vaccinations est illustrée à la figure 4.5 car ce cas de sous-fonction est également pertinent pour l'achat d'une jument.

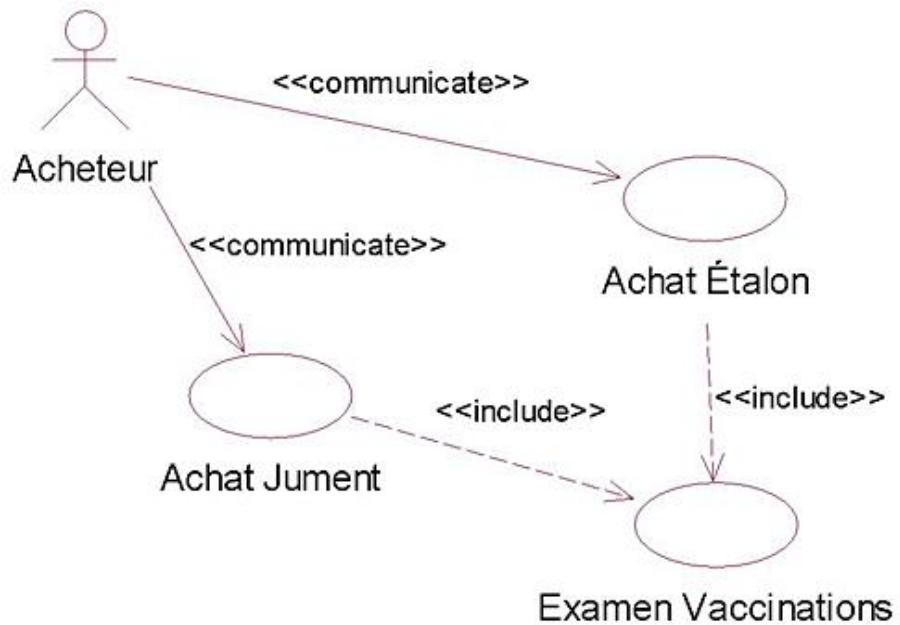


Figure 4.5 - Mise en commun d'un cas d'utilisation inclus

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 4.6, 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.

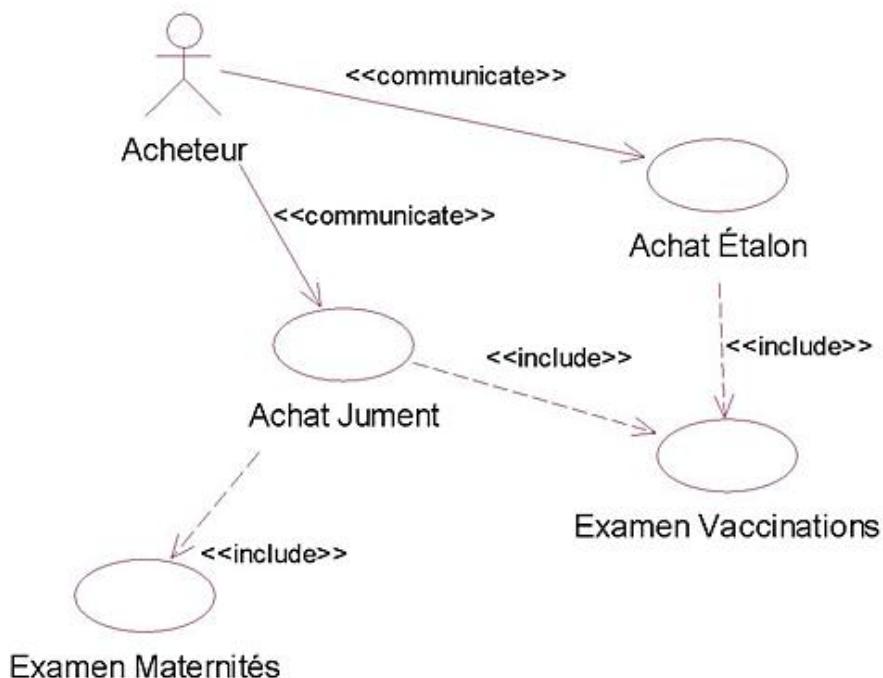


Figure 4.6 - Décomposition d'un cas d'utilisation par inclusion

2. 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 ses vérifications (voir figure 4.7).

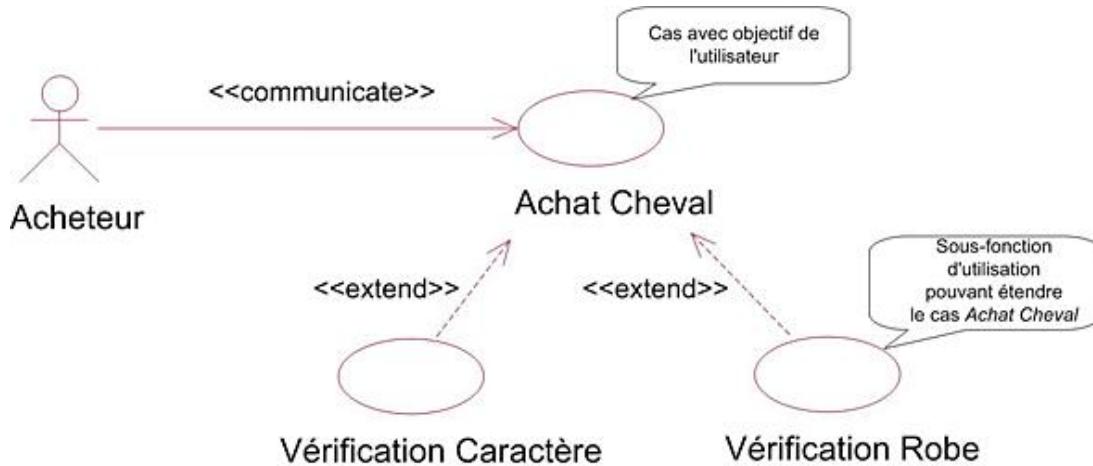


Figure 4.7 - Extension d'un cas d'utilisation

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 4.8). D'autre part, les cas d'utilisation de la vérification du caractère et de la vérification de la robe sont partagés.

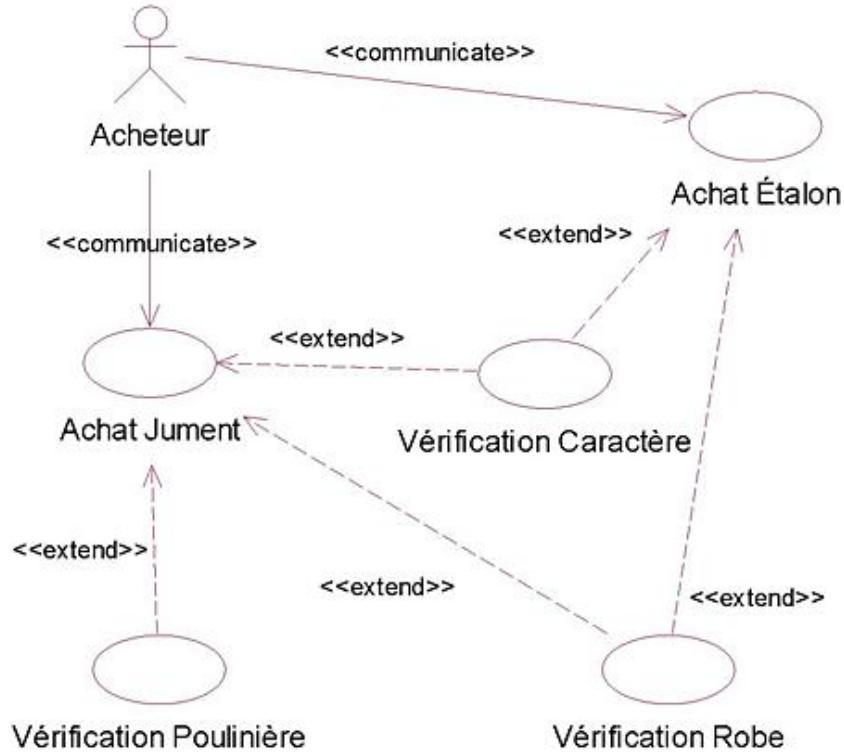


Figure 4.8 - Extensions partagées de cas d'utilisation

3. La spécialisation et la généralisation des cas d'utilisation

Comme nous l'avons vu dans le chapitre Les concepts de l'approche par objets pour les classes d'objet, il est également possible de spécialiser un cas d'utilisation en un autre. On obtient ainsi un sous-cas d'utilisation.

Comme pour les classes, le sous-cas hérite du comportement du sur-cas d'utilisation. Un sous-cas d'utilisation hérite également des relations de communication, d'inclusion et d'extension du sur-cas.

Souvent, le sur-cas d'utilisation est abstrait, c'est-à-dire qu'il correspond à un comportement partiel complété dans les sous-cas d'utilisation.

Un sous-cas d'utilisation a le même niveau que son sur-cas. Si le sur-cas est un cas avec objectif utilisateur, il en va de même pour le sous-cas. Si le sur-cas est un cas de sous-fonction, le sous-cas est lui aussi une sous-fonction.

Dans le diagramme des cas d'utilisation, la relation de spécialisation est représentée par une flèche pleine de spécialisation identique à celle qui relie les sous-classes aux superclasses. Le nom d'un cas d'utilisation abstrait est écrit en italique (ou accompagné du stéréotype «*abstract*»).

Exemple

Le cas d'utilisation d'achat d'un cheval est spécialisé en deux sous-cas : l'achat d'une jument ou d'un étalon. Ce cas est un cas abstrait et son nom apparaît en italiques. La figure 4.9 illustre cette spécialisation.

Les cas d'utilisation d'achat de la jument et d'achat de l'étalon sont des cas d'utilisation avec objectif utilisateur et communiquent avec l'acheteur. En effet, la relation de communication qui existe entre le cas d'utilisation d'achat du cheval et Acheteur est héritée dans les deux sous-cas d'utilisation.

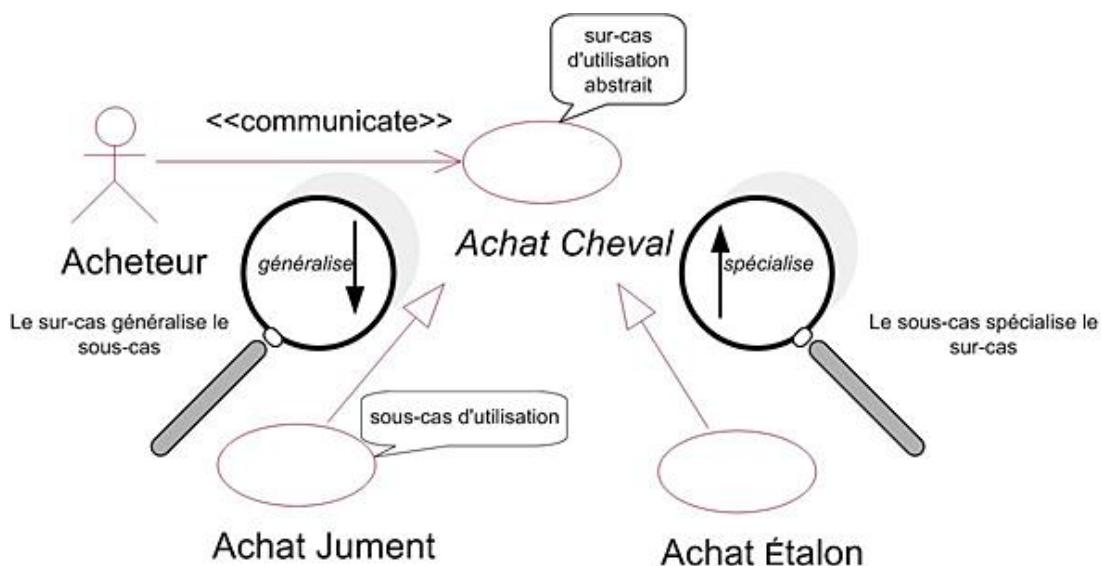


Figure 4.9 - Spécialisation d'un cas d'utilisation

Exemple

Les relations d'extension concernant les différentes inclusions et extensions de vérification peuvent être factorisées au niveau du cas abstrait. Elles sont alors héritées dans les sous-cas à l'image de la relation de communication dans l'exemple précédent (voir figure 4.10).

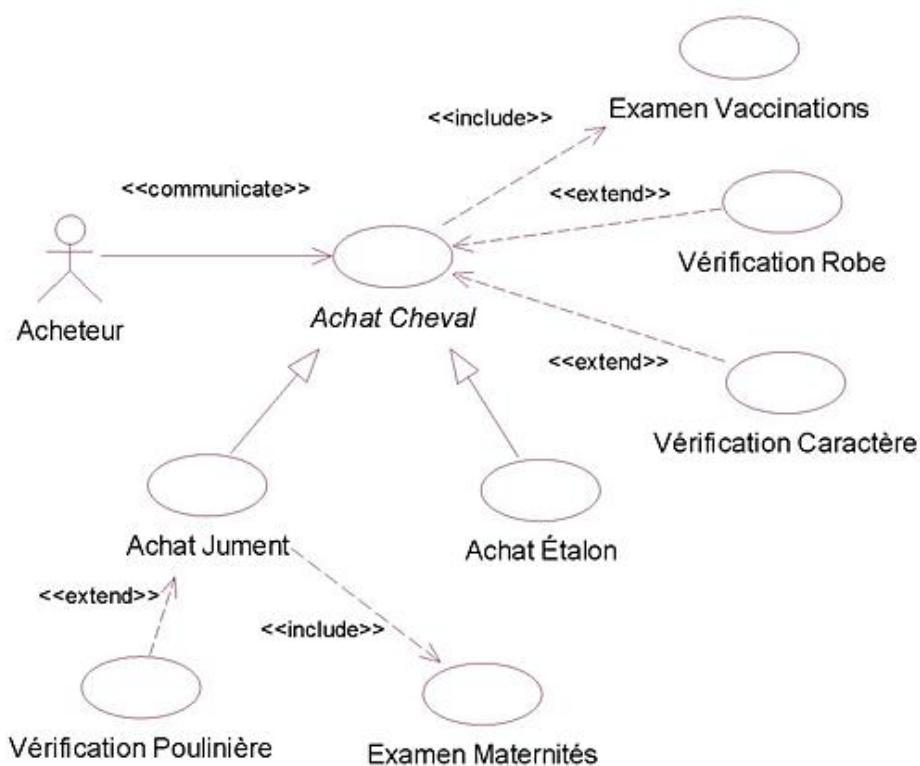


Figure 4.10 - Factorisation d'inclusion et d'extension

La représentation textuelle des cas d'utilisation

La représentation textuelle des cas d'utilisation n'est pas spécifiée dans UML. Elle est cependant couramment utilisée, c'est pourquoi nous l'avons introduite dans cet ouvrage.

Cette représentation sous forme textuelle des cas d'utilisation donne une description de leurs comportements, de leurs actions et réactions. Le contenu de cette représentation textuelle est la suivante :

- le nom du cas d'utilisation ;
- l'acteur primaire ;
- le système concerné par le cas d'utilisation ;
- les intervenants (ensemble des acteurs) ;
- le niveau du cas d'utilisation pouvant être soit :
 - un objectif utilisateur ;
 - ou une sous-fonction ;
- les préconditions qui sont les conditions à remplir pour que le cas d'utilisation puisse être exécuté ;
- les opérations du scénario principal ;
- les extensions.

Cas d'utilisation	Nom du cas d'utilisation
Acteur primaire	Nom de l'acteur primaire
Système	Nom du système
Intervenants	Nom des intervenants
Niveau	Objectif utilisateur ou sous-fonction
Préconditions	Conditions devant être remplies pour exécuter le cas d'utilisation
Opérations	
1	Opération 1
2	Opération 2
3	Opération 3
4	Opération 4
5	Opération 5
Extensions	
1.A	Condition d'application de l'extension A sur l'opération 1
1.A.1	Opération 1 de l'extension A sur l'opération 1

1.A.2	Opération 2 de l'extension A sur l'opération 1
1.B	Condition d'application de l'extension B sur l'opération 1
1.B.1	Opération 1 de l'extension B sur l'opération 1
4.A	Condition d'application de l'extension A sur l'opération 4
4.A.1	Opération 1 de l'extension A sur l'opération 4

Exemple

Le cas d'utilisation d'achat d'une jument est illustré ci-après. Chaque extension est numérotée par la ligne de l'opération à laquelle elle s'applique suivie d'une lettre qui permet de distinguer chaque extension d'une même ligne. Ensuite, chaque opération d'une extension est numérotée, de la même façon que les opérations du scénario principal.

Cas d'utilisation	Achat d'une jument
Acteur primaire	Acheteur
Système	Élevage de chevaux
Intervenants	Acheteur, Haras nationaux
Niveau	Objectif utilisateur
Précondition	La jument est à vendre
Opérations	
1	Choisir la jument
2	Vérifier les vaccinations
3	Examiner les maternités
4	Recevoir une proposition de prix
5	Évaluer la proposition de prix
6	Payer le prix de la jument
7	Remplir les papiers de vente
8	Enregistrer la vente auprès des haras nationaux
9	Aller chercher la jument
10	Transporter la jument
Extensions	
2.A	Les vaccinations conviennent-elles ?
2.A.1	Si oui, continuer
2.A.2	Si non, abandonner
3.A	L'examen des maternités convient-il ?

3.A.1	Si oui, continuer
3.A.2	Si non, abandonner
5.A	Le prix convient-il ?
5.A.1	Si oui, continuer
5.A.2	Si non, négocier le tarif et réexécuter l'étape 5

Conclusion

Les cas d'utilisation servent à :

- exprimer les exigences fonctionnelles conférées au système par les utilisateurs lors de la rédaction du cahier des charges ;
- vérifier que le système répond à ces exigences lors de la livraison ;
- déterminer les frontières du système ;
- écrire la documentation du système ;
- construire les jeux de test.

Les cas d'utilisation offrent une technique de représentation qui convient au dialogue avec l'utilisateur car son formalisme reste proche du langage naturel. Il est conseillé d'y adjoindre un lexique pour éviter les risques de confusion.

Nous étudierons par la suite comment découvrir les objets en utilisant les diagrammes de séquence associés aux cas d'utilisation.

Exercices

1. L'hippodrome

Un hippodrome offre à ses clients la possibilité de suivre les courses et de parier.

Quels sont les acteurs qui interagissent avec ces services ?

Construire le diagramme des cas d'utilisation.

2. Le club équestre

Un club équestre offre les prestations d'hébergement des chevaux, de cours d'équitation, de balades. Seuls les adhérents ont accès aux cours et aux hébergements. Les autres clients ont la possibilité de faire des balades et d'adhérer.

Quels sont les acteurs qui interagissent avec ces services ?

Construire le diagramme des cas d'utilisation.

3. Le manège de chevaux de bois

Un manège de chevaux de bois offre à ses clients la possibilité de faire un tour moyennant paiement.

Quels sont les acteurs liés à ce service ?

Construire le diagramme des cas d'utilisation.

Donner la représentation textuelle correspondant au diagramme.

Introduction

Ce chapitre a pour objectif de vous faire découvrir comment UML représente les interactions entre les objets. Nous avons découvert au chapitre Les concepts de l'approche par objets que les objets d'un système possèdent leur propre comportement et interagissent entre eux afin de doter le système de sa dynamique globale. Nous avons étudié au chapitre La modélisation des exigences la façon dont les cas d'utilisation représentent les actions et réactions entre un acteur externe et le système. Du point de vue de la modélisation, ces deux types d'interactions se distinguent par leur différence interne/externe mais nullement par leur nature.

UML propose deux diagrammes pour répondre à ce besoin de représentation des interactions entre objets :

- Le diagramme de séquence se focalise sur les aspects temporels.
- Le diagramme de communication se focalise sur la représentation spatiale.

Dans ce chapitre, nous allons étudier ces deux diagrammes. Nous examinerons ensuite comment découvrir progressivement les objets composant un système. Cette découverte sera basée sur les interactions entre objets intervenant dans les cas d'utilisation du système. Pour représenter ces interactions, nous ferons le choix du diagramme de séquence, cette option ayant très souvent la faveur des personnes chargées de la modélisation d'un projet.

 Le diagramme de communication porte ce nom depuis UML 2. En UML 1, il s'appelait diagramme de collaboration.

Le diagramme de séquence

1. Définition

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 flux de données échangées lors des envois de message.

- Pour interagir entre eux, les objets s'envoient des messages. Lors de la réception d'un message, un objet devient actif et exécute la méthode de même nom. Un envoi de message est donc un appel de méthode.

2. La ligne de vie d'un objet

Comme il représente la dynamique du système, le diagramme de séquence fait entrer en action les instances des classes intervenant dans la réalisation de la sous-fonction qui lui est liée. À chaque instance est associée une ligne de vie qui montre ses actions et réactions, ainsi que les périodes pendant lesquelles elle est active, c'est-à-dire où elle exécute l'une de ses méthodes.

La représentation graphique de la ligne de vie est illustrée à la figure 5.1.

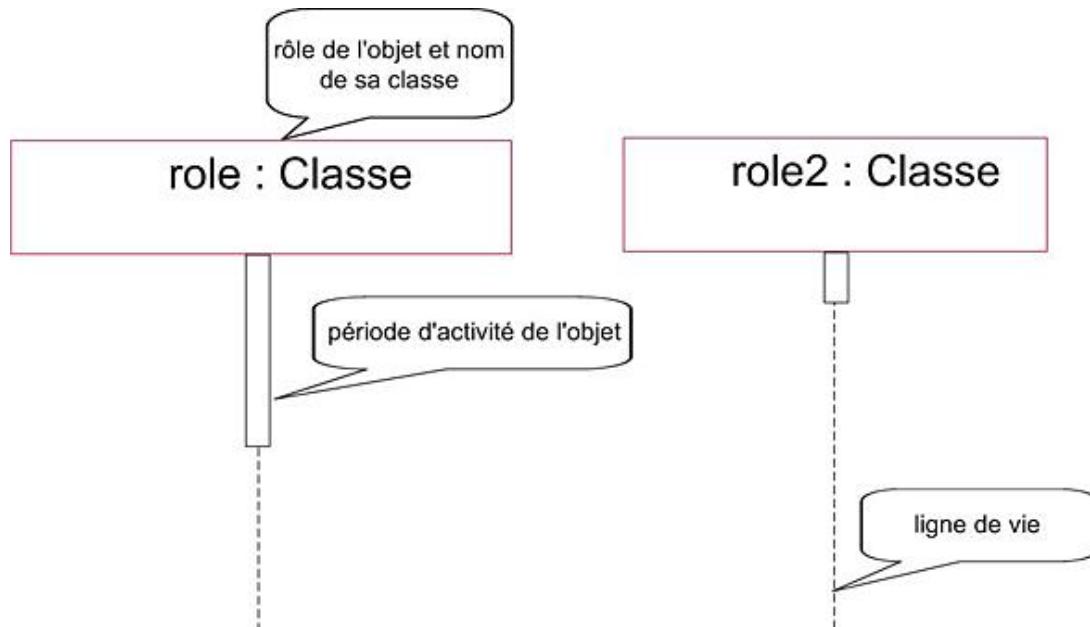


Figure 5.1 - Lignes de vie

- La notation "role : Classe" représente le rôle d'une instance suivi du nom de sa classe. Dans cet ouvrage, par souci de simplification, nous considérons que le rôle de l'instance correspond à son nom, comme c'était le cas en UML 1. Le rôle de l'instance est optionnel si une seule instance de cette classe participe au diagramme de séquence. Le nom de la classe peut également être omis dans le cas d'une étape préliminaire de la modélisation mais il doit être spécifié dès que possible.

- Un diagramme de séquence contient plusieurs lignes de vie car il traite des interactions entre plusieurs objets.

3. L'envoi de message

Les envois de message sont représentés par des flèches horizontales reliant la ligne de vie de l'objet émetteur à la ligne de vie de l'objet destinataire (voir figure 5.2).



Figure 5.2 - Envoi d'un message

Dans la figure 5.2, l'objet de gauche envoie un message à l'objet de droite. Ce message donne lieu à l'exécution de la méthode *message* de l'objet de droite, ce qui provoque son activation.

Les messages sont numérotés séquentiellement, à partir de un. Si un message est envoyé alors que le traitement du précédent n'est pas terminé, il est possible d'utiliser une numération composée (voir figure 5.3) où l'envoi du message 2 intervient pendant l'exécution du message 1.

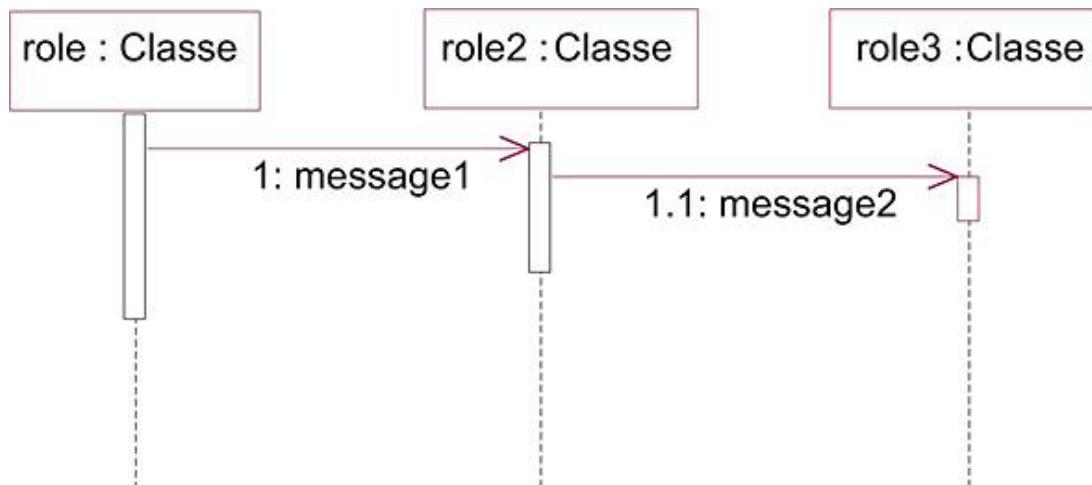


Figure 5.3 - Numérotation des messages

➤ La numérotation des messages n'est pas obligatoire. Elle reste toutefois pratique pour montrer les activations imbriquées.

La transmission d'information est également possible ; elle est représentée par des paramètres transmis avec le message (voir figure 5.4).



Figure 5.4 - Transmission de données lors de l'envoi d'un message

Il existe différents types d'envois de message. La figure 5.5 en fournit une explication graphique.

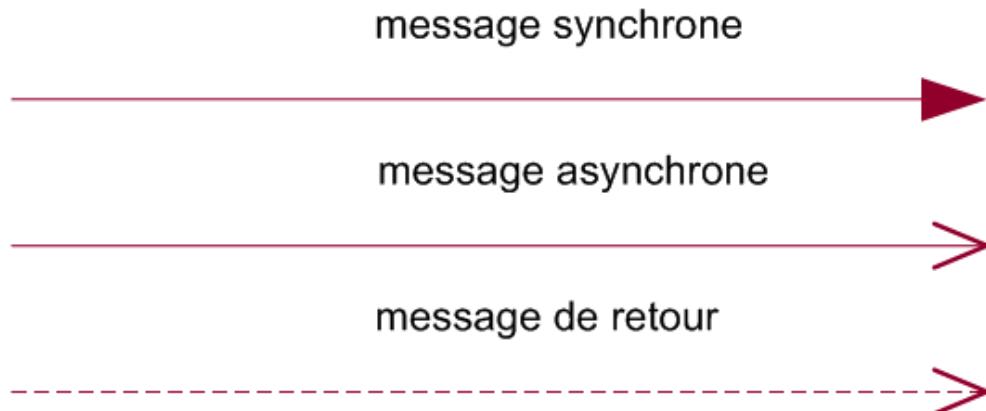


Figure 5.5 - Les différents types de messages

Le message synchrone est le plus fréquemment utilisé. Son emploi signifie que l'expéditeur du message attend que l'activation de la méthode invoquée chez le destinataire soit terminée avant de continuer son activité.

Dans le cas du message asynchrone, l'expéditeur n'attend pas la fin de l'activation de la méthode invoquée chez le destinataire. Ceci se produit lors de la modélisation d'un système où les objets peuvent fonctionner en parallèle (cas des systèmes multi-thread où les traitements sont effectués en parallèle).

Exemple

Un cavalier donne un ordre à son cheval puis un second ordre sans attendre la fin de l'exécution du précédent. Le premier ordre constitue un exemple d'envoi de message asynchrone.

-
- En UML 1, dans la représentation d'un message asynchrone, une demi-flèche supérieure était utilisée. Avec UML 2, c'est une flèche complète qui est employée.

Le message de retour de l'invocation d'une méthode n'est pas systématique, toutes les méthodes ne retournant pas un résultat.

Un objet peut envoyer un message à lui-même. La représentation d'un tel message est illustrée à la figure 5.6.

role : Classe

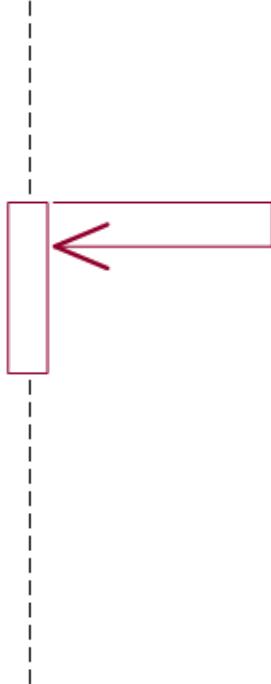


Figure 5.6 - Envoi d'un message à soi-même

4. La création et la destruction d'objets

Le diagramme de séquence décrivant la dynamique d'un système, celle-ci contient fréquemment des créations et des destructions d'objets.

La création d'objet est représentée par un message spécifique qui donne lieu au début de la ligne de vie du nouvel objet.

La destruction d'objet est un message envoyé à un objet existant et qui donne lieu à la fin de sa ligne de vie. Il est représenté par une croix.

Ces deux messages sont illustrés à la figure 5.7.

role : Classe

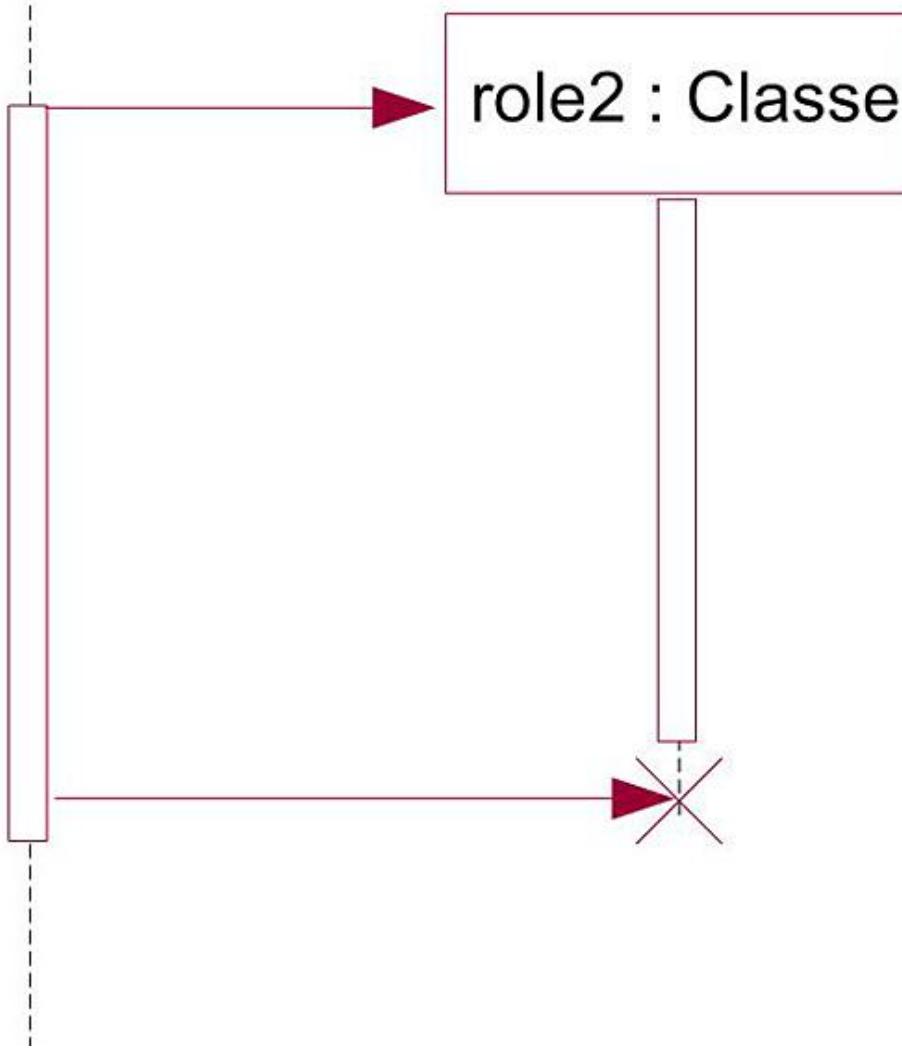


Figure 5.7 - Messages de création et de destruction d'un objet

5. La description de la dynamique

À partir des différents éléments introduits précédemment, il est maintenant possible de construire l'intégralité d'un diagramme de séquence et de décrire la dynamique d'un petit système ou une sous-fonction d'un système plus important.

Exemple

La figure 5.8 représente un scénario d'achat d'une jument déjà étudié au chapitre *La modélisation des exigences*. Il n'y a aucune alternative ; il s'agit donc bien d'un scénario. Nous verrons par la suite comment introduire les alternatives et les boucles.

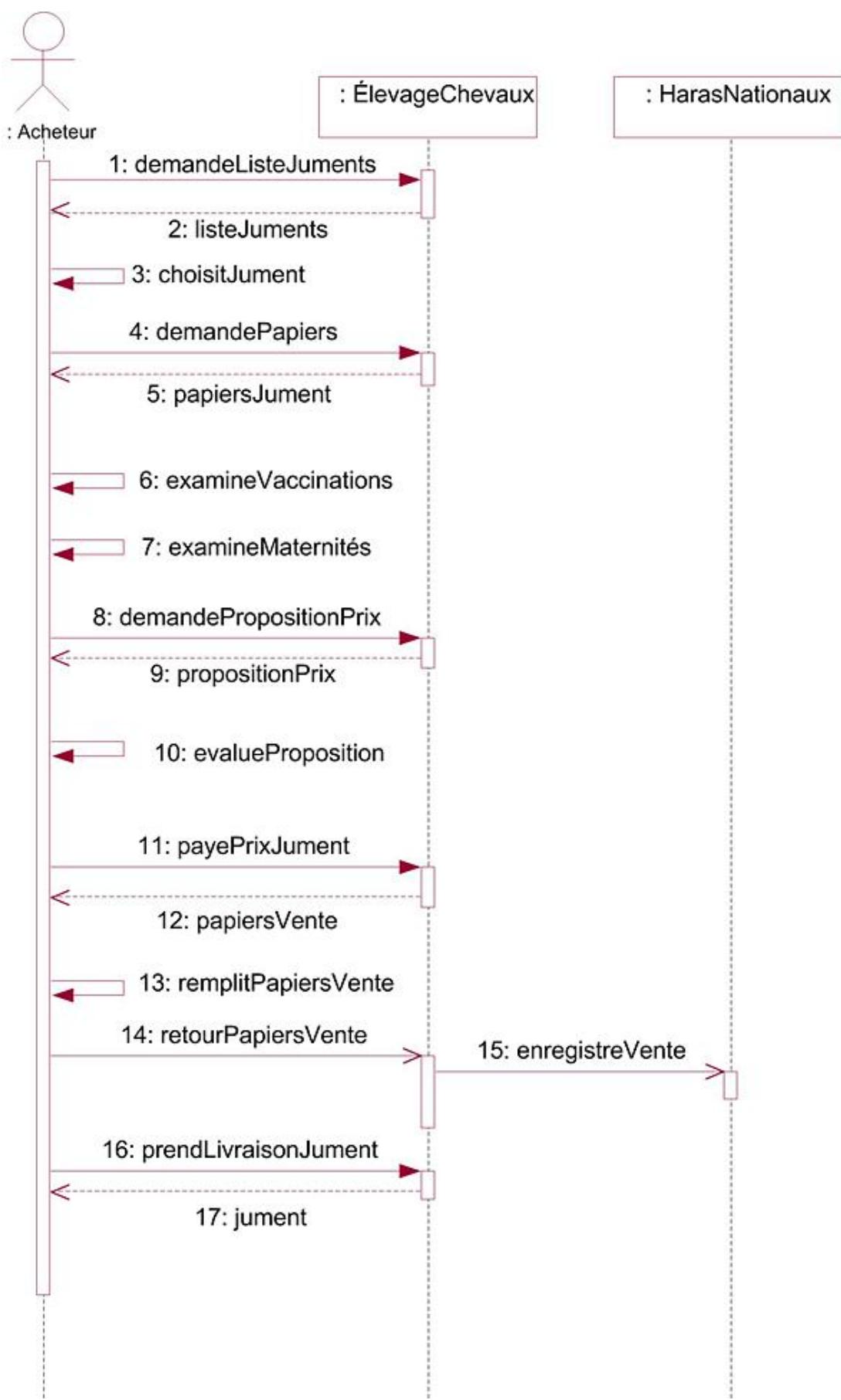


Figure 5.8 - Exemple de diagramme de séquence : la représentation d'un scénario d'achat d'une jument

Les cadres d'interaction (UML 2)

Jusqu'ici, les constructions introduites pour l'écriture des diagrammes de séquence sont celles d'UML 1. Les diagrammes ainsi construits décrivent des scénarios. Par conséquent, pour représenter tous les scénarios de la dynamique d'un système, il convient d'écrire autant de diagrammes.

UML 2 généralise les diagrammes de séquence pour y introduire les cadres d'interaction. Cette extension importante offre le support des alternatives et des boucles et confère au diagramme de séquence le statut d'un véritable modèle des interactions.

1. La notion de cadre d'interaction

Un cadre d'interaction est une partie du diagramme de séquence associé à une étiquette. Elle contient un opérateur qui en détermine la modalité d'exécution. Les principales modalités sont le branchement conditionnel et la boucle.

2. L'alternative

L'alternative s'obtient en utilisant l'opérateur opt suivi d'une condition de test. Si la condition est vérifiée, le contenu du cadre est exécuté.



Figure 5.9 - Cadre d'interaction d'alternative

Il existe un autre opérateur pour l'alternative. Nommé alt, il est suivi de plusieurs conditions de test puis du mot clé else. Le cadre est alors scindé en plusieurs parties dont le contenu n'est exécuté que si la condition associée est remplie. Le contenu de la dernière partie est associé au mot clé else (sinon). Il est exécuté uniquement si aucune des conditions précédentes n'est vérifiée.

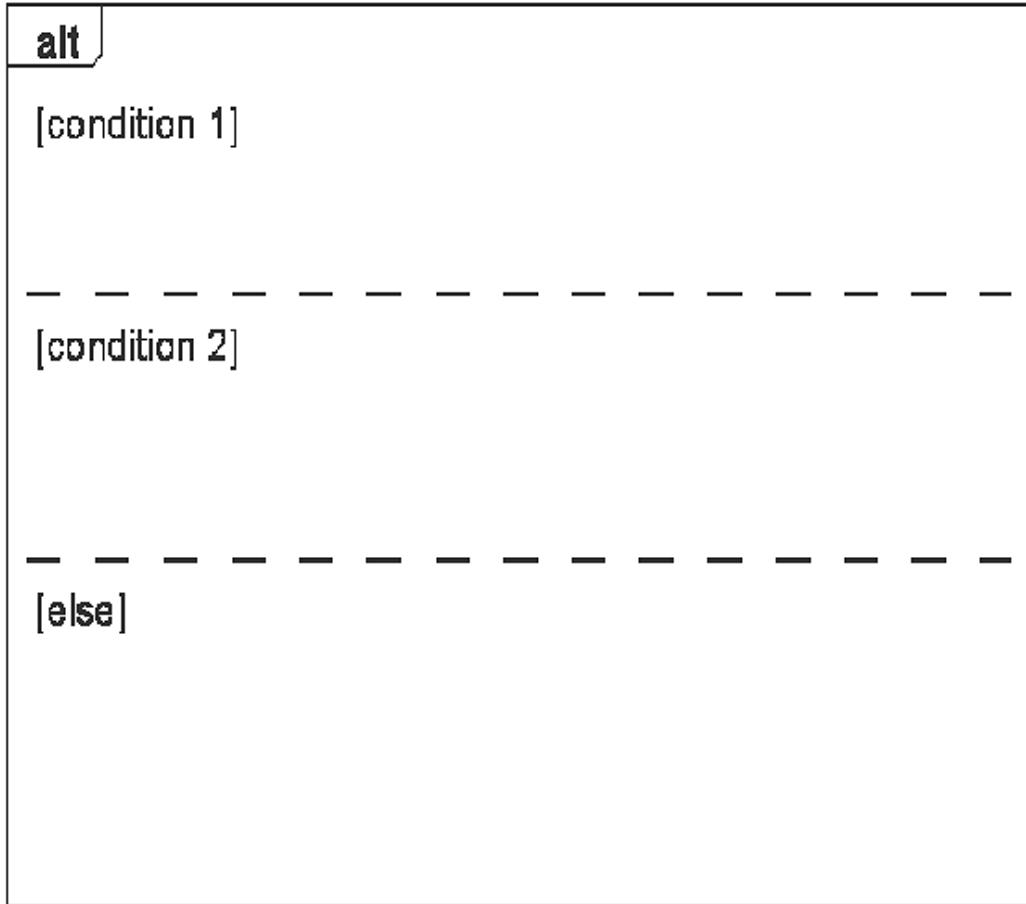


Figure 5.10 - Cadre d'interaction basé sur l'opérateur alt

3. La boucle

La boucle est réalisée par l'opérateur *loop* suivi des paramètres *min*, *max* et d'une condition de test. Le contenu du cadre est exécuté *min* fois, puis tant que la condition de test est vérifiée et tant que le nombre maximal d'exécutions de la boucle ne dépasse pas *max*. Chaque paramètre est optionnel.

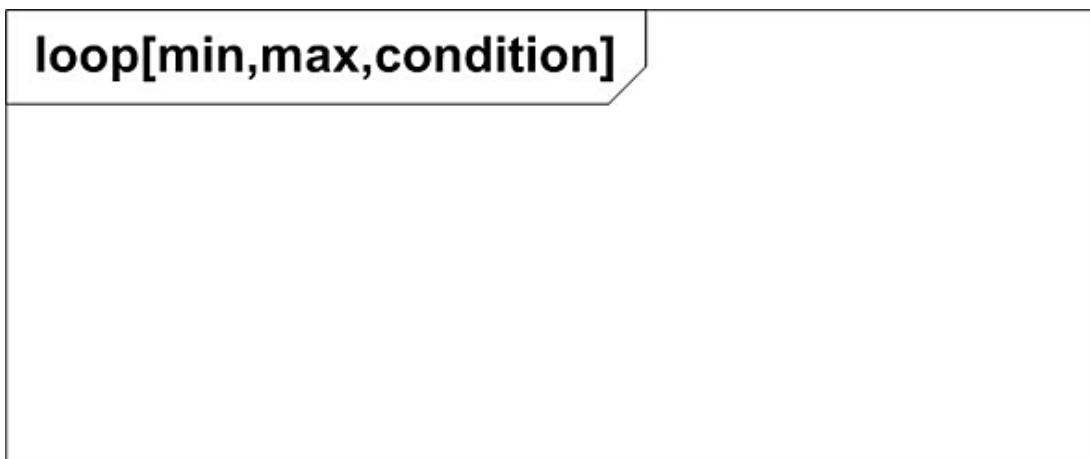


Figure 5.11 - Cadre d'interaction de boucle

Exemple

Pour franchir un obstacle, un cavalier peut s'y prendre à plusieurs reprises, sans toutefois dépasser deux refus.

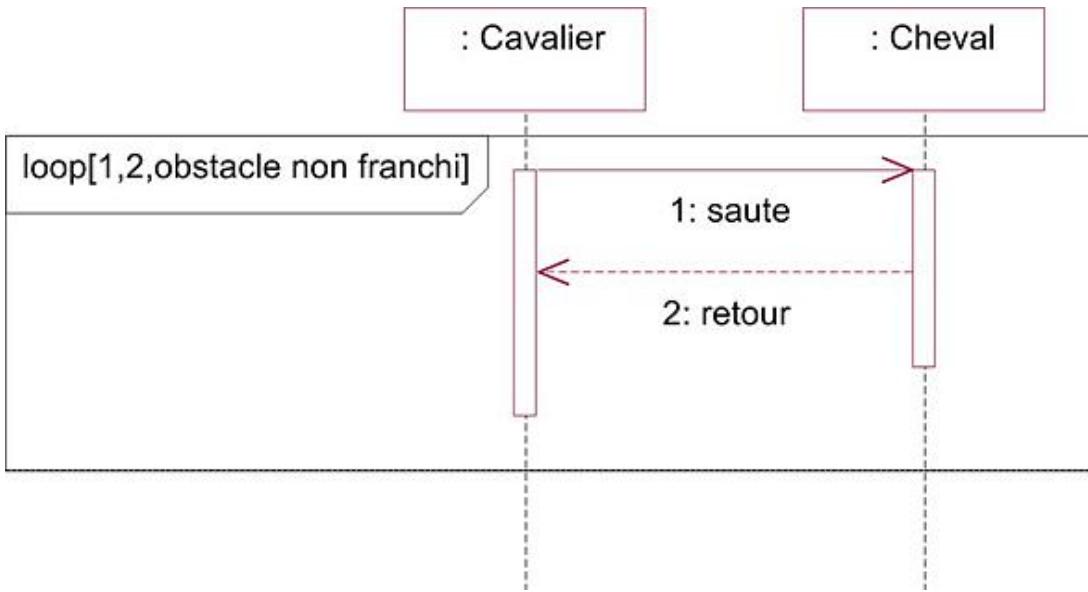


Figure 5.12 - Exemple de boucle

4. Utilisation des cadres d'interaction

À partir des différents éléments introduits jusqu'ici, il est maintenant possible de décrire de façon plus générale la dynamique du système.

Exemple

La figure 5.13 représente le cas d'utilisation d'achat d'une jument. À la différence de la figure 5.8, les alternatives et boucles introduites dans le cas d'utilisation sont prises en compte. Notamment, le diagramme de séquence n'est exécuté jusqu'au bout que si les vaccinations et maternités sont validées par l'acheteur. Par ailleurs, la boucle de négociation du prix de vente est également représentée.

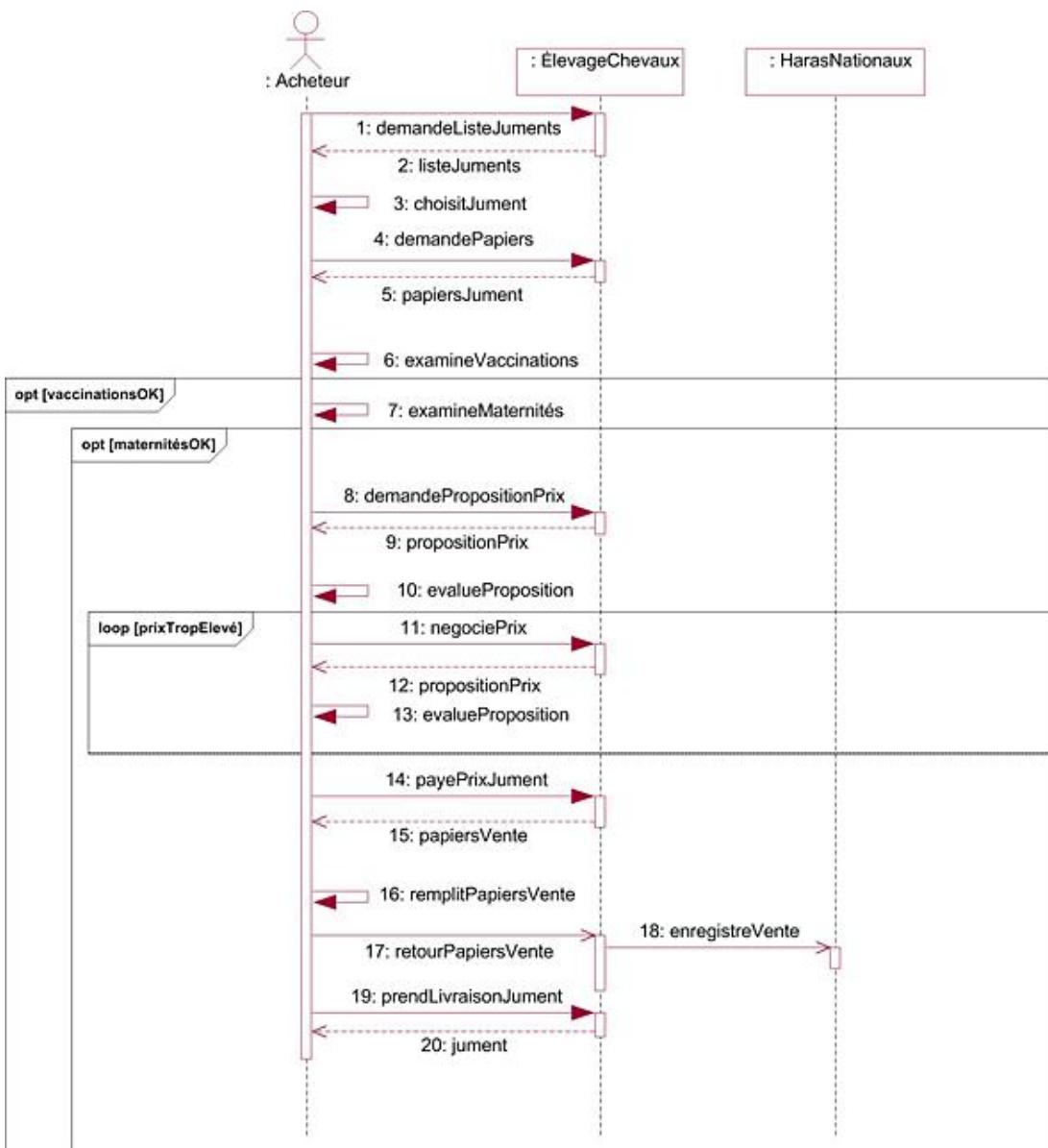


Figure 5.13 - Exemple de diagramme de séquence : la représentation du cas d'utilisation d'achat d'une jument

Le diagramme de communication

Le diagramme de communication est une alternative au diagramme de séquence. Il se focalise sur une représentation spatiale des objets.

Chaque objet intervient dans ce diagramme de la même façon que dans le diagramme de séquence. Il n'est pas associé à une ligne de vie mais relié graphiquement aux objets avec lesquels il interagit.

Les envois de messages sont placés le long des liens interobjets. Les messages sont obligatoirement numérotés, la numérotation composée étudiée dans le cadre des diagrammes de séquence pouvant également être employée.

La figure 5.14 illustre le diagramme de communication qui correspond au diagramme de séquence de la figure 5.3.

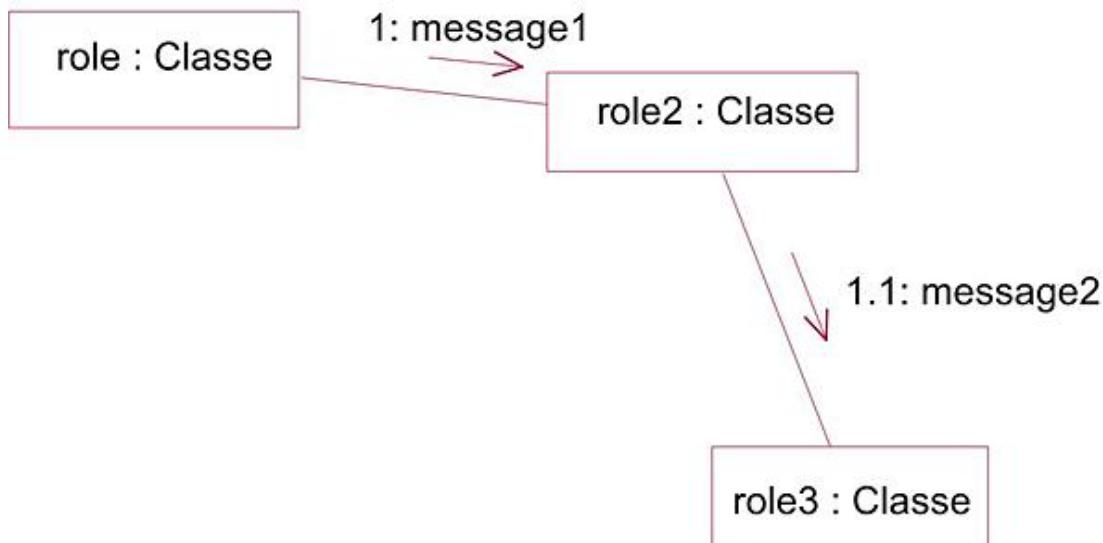


Figure 5.14 - Diagramme de communication

Il est également possible de transmettre des informations lors de l'envoi, de la même façon que dans le cas du diagramme de séquence.

Exemple

Dans un troupeau de chevaux, il existe une jument dominante qui est responsable de l'éducation de tous les poulains. Elle peut passer le relais à une autre jument pour la surveillance d'un poulain particulier.

Dans l'exemple de la figure 5.15, la jument dominante délègue la surveillance du poulain Espiègle à une autre jument, qui lui donne un ordre auquel il refuse d'obéir. Il est donc puni.

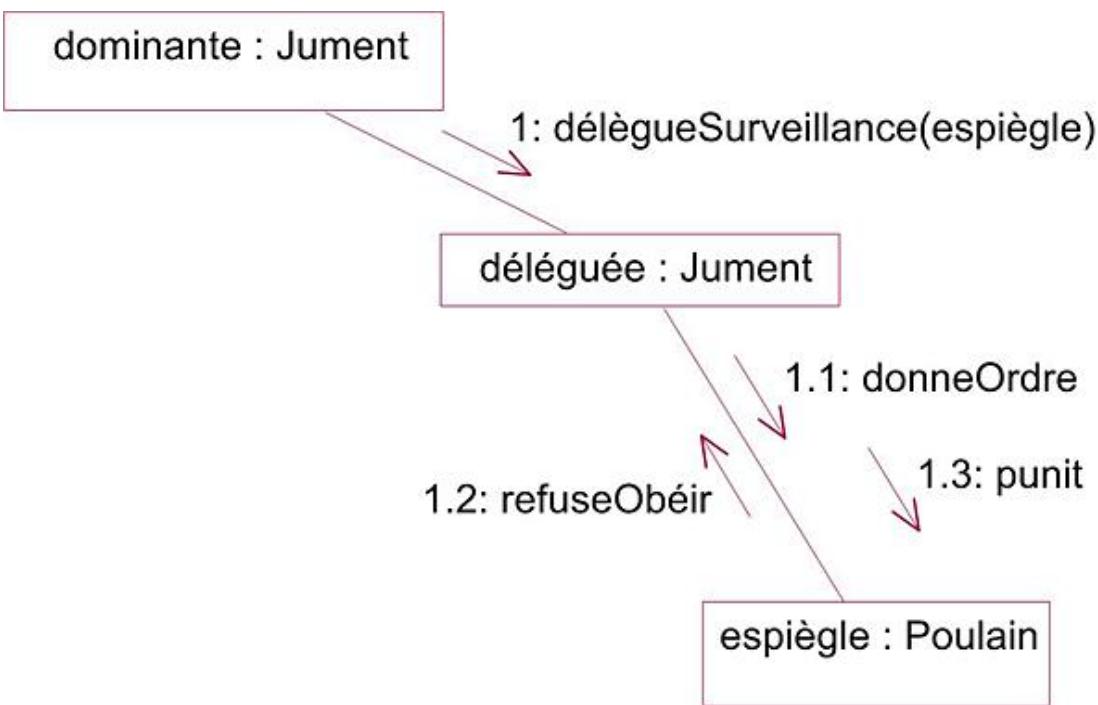


Figure 5.15 - Exemple de diagramme de communication : la surveillance du poulain Espiègle

Comme il n'existe pas l'équivalent des cadres d'interaction, UML propose des mécanismes de test et de boucle au niveau de l'envoi des messages.

Les mécanismes de test sont réalisés par une condition spécifiée entre crochets après le numéro du message, ce qui donne la syntaxe suivante :

Numéro[condition] : message

Il existe deux mécanismes de boucle :

- Le premier est basé sur une condition. La boucle s'exécute tant que la condition est vraie. Sa syntaxe est la suivante : Numéro*[condition]: message
- Le second est basé sur une variable de boucle dont les limites inférieures et supérieures sont spécifiées. Sa syntaxe est la suivante : Numéro*[variableBoucle=limiteInf..limiteSup]: message

Exemple

Quand un poulain atteint l'âge de deux ans, il est chassé du troupeau par l'étalon.

La figure 5.16 présente ce message pour le poulain Espiègle.

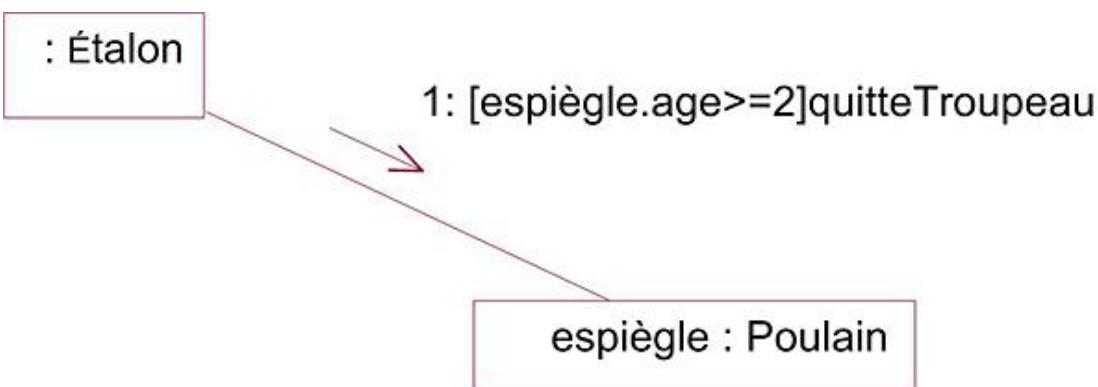


Figure 5.16 - Exemple de condition : le poulain Espiègle doit quitter le troupeau à l'âge de deux ans

➤ L'âge est un attribut du poulain Espiègle. On y accède par la syntaxe : nomObjet.nomAttribut. Cette syntaxe est recommandée pour l'accès aux attributs et aux méthodes d'un objet dans une condition.

Découvrir les objets du système

Nous avons vu qu'il est aisé de représenter un cas d'utilisation par un diagramme de séquence, surtout depuis l'introduction des cadres d'interaction d'UML 2.

Dans ce diagramme, le système est représenté sous la forme d'un objet et les interactions avec l'extérieur se produisent le long de sa ligne de vie.

Pour découvrir les objets du système à partir d'un cas d'utilisation, une première phase consiste à se demander à quels objets du système sont destinés les messages provenant de l'extérieur.

Leur détermination constitue une première étape de décomposition du système en objets.

Exemple

Dans l'exemple du cas d'utilisation d'achat d'une jument, les interactions entre l'acheteur et l'élevage de chevaux peuvent être décomposées en interactions entre d'une part l'acheteur et d'autre part le directeur, le comptable ou le palefrenier de l'élevage (voir figure 5.17).

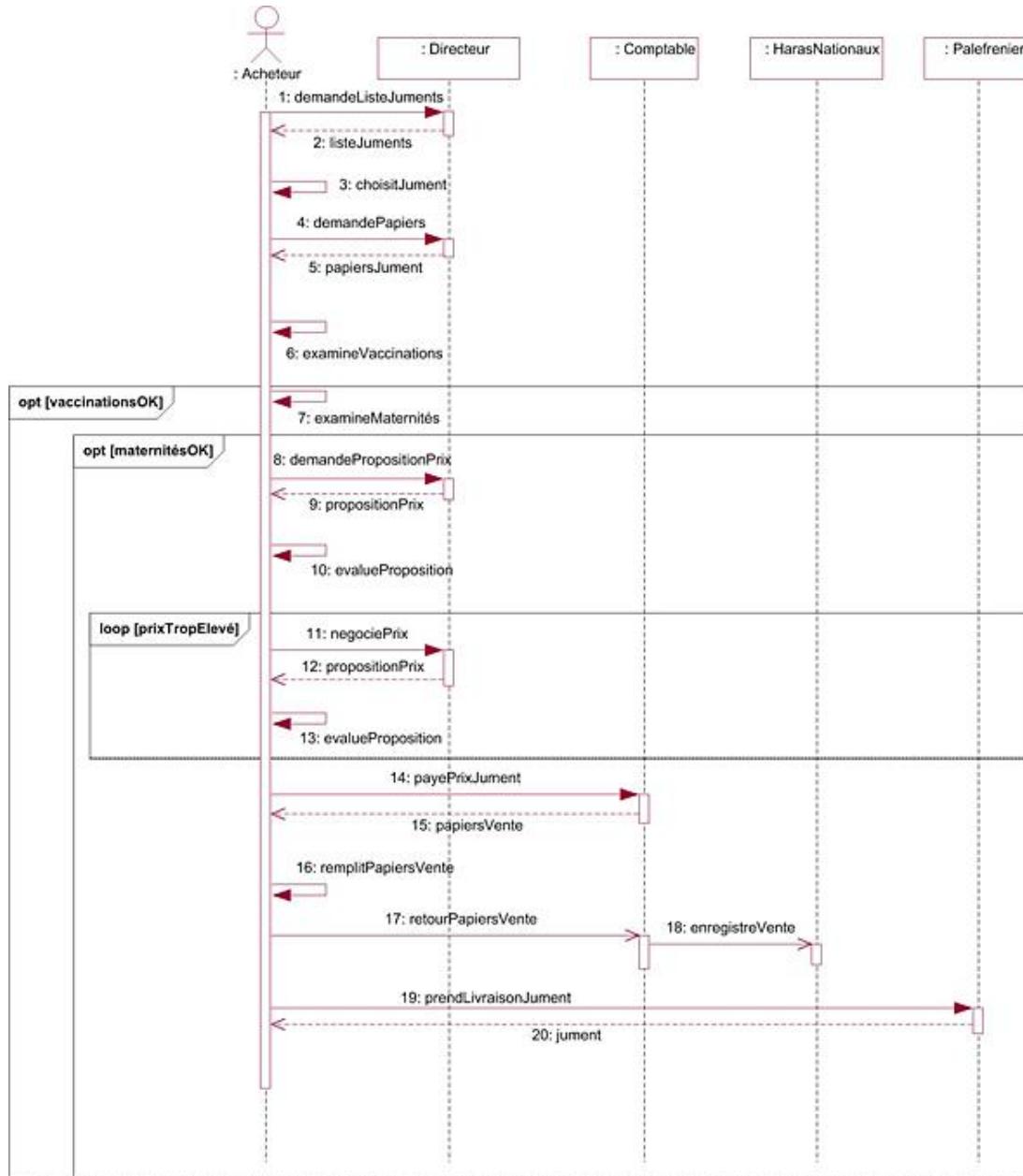


Figure 5.17 - Premier enrichissement du diagramme de séquence d'achat d'une jument

Dans un second temps, les messages reçus par les objets du système sont décomposés progressivement ; on continue ainsi de découvrir au fur et à mesure les objets du système. En effet, la décomposition des messages oblige à utiliser de nouveaux objets qui répondent aux fonctionnalités requises.

Cette décomposition s'accompagne fréquemment d'un enrichissement de la description des messages au niveau de la transmission des informations. En effet, le traitement de ces informations implique souvent l'utilisation de nouveaux objets.

Exemple

Lorsque le directeur reçoit la demande des papiers de la jument, il fait appel à la base de données de l'élevage pour les retrouver. Le diagramme de séquence correspondant est enrichi en conséquence à la figure 5.18 (vue partielle correspondant à la recherche des papiers). Au sein de ce diagramme, la jument choisie est dorénavant transmise comme paramètre ; ainsi l'on peut retrouver ses papiers dans la base de données de l'élevage.

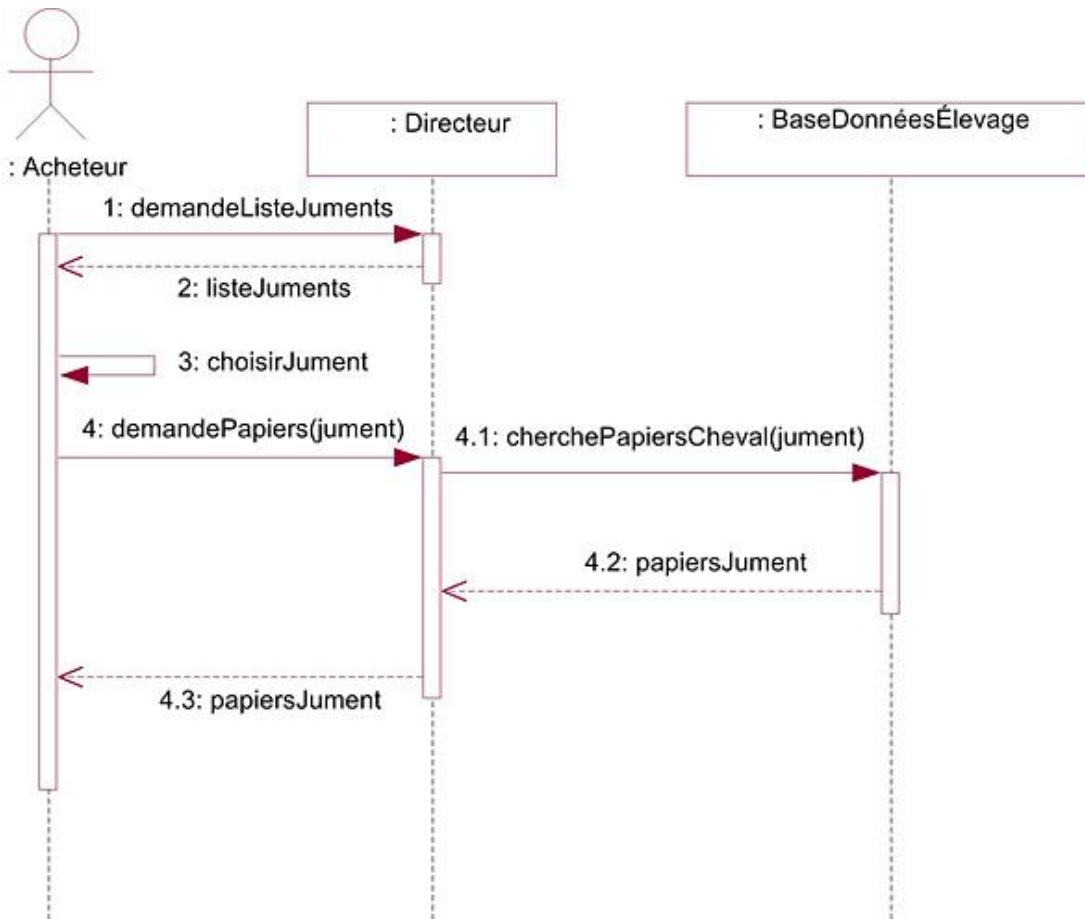


Figure 5.18 - Deuxième enrichissement du diagramme de séquence d'achat d'une jument (vue partielle)

Exemple

Lorsque le comptable reçoit le paiement, il crée les papiers de la vente avant de les renvoyer à l'acheteur. Cette décomposition introduit un nouvel objet du système : ces papiers.

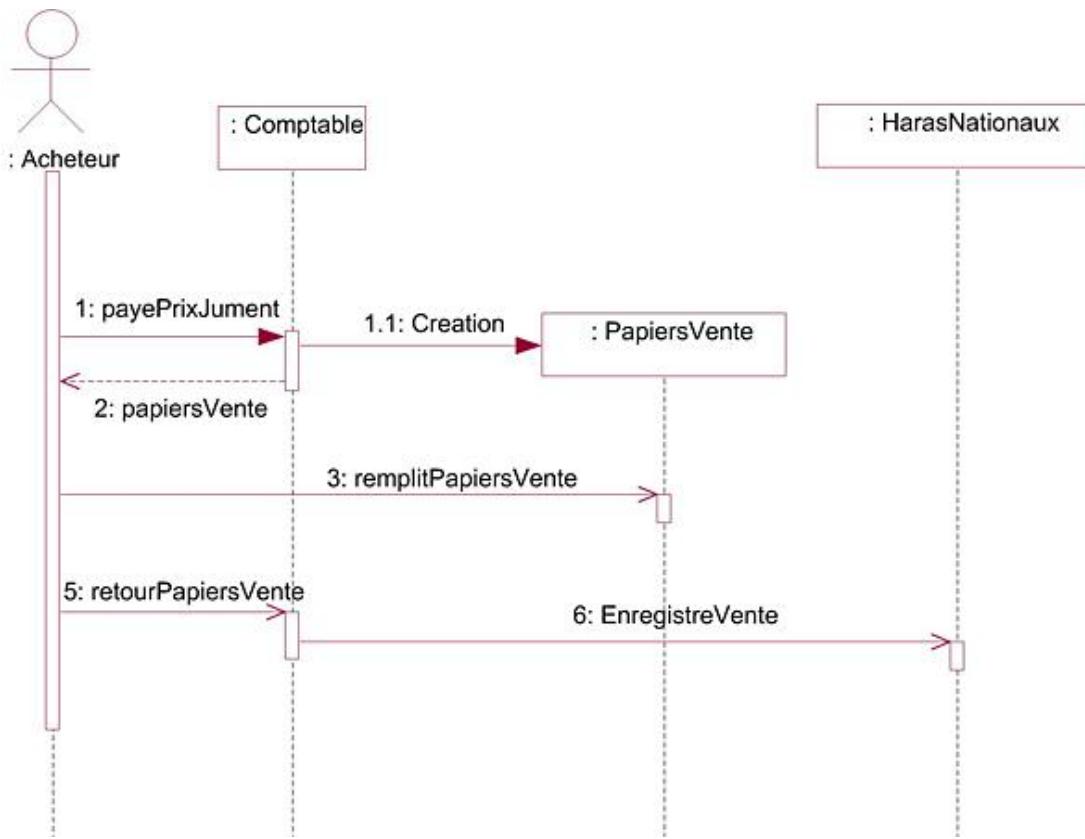


Figure 5.19 - Troisième enrichissement du diagramme de séquence d'achat d'une jument (vue partielle)

Ce processus de décomposition des messages est itératif et doit être poursuivi jusqu'à l'obtention d'objets de taille suffisamment fine.

➤ On emploie le plus souvent le mot grain pour évoquer la taille d'un objet. En effet, il existe toute une panoplie de grains allant du plus fin au plus gros. Le système pris comme un seul objet est un objet de gros grain ou de granularité importante. À l'opposé, la base de données de l'élevage est déjà un objet de granularité plus fine. En son sein, une table ou une donnée seront des objets de granularité encore plus fine. Cette notion est relative. C'est à la personne en charge de la modélisation de déterminer le niveau de granularité des objets qu'elle désire obtenir.

Conclusion

Les diagrammes de séquence et de communication sont importants pour les raisons suivantes :

- illustrer et vérifier le comportement d'un ensemble d'objets (système ou sous-système) ;
- aider à la découverte des objets du système ;
- aider à la découverte des méthodes des objets.

Depuis l'introduction des cadres d'interaction, les diagrammes de séquence peuvent être utilisés pour décrire les cas d'utilisation.

Les diagrammes de séquence mettent en avant les aspects temporels tandis que les diagrammes de communication montrent les liaisons entre les classes.

Exercices

1. L'hippodrome

Construire le diagramme de séquence d'achat d'un billet pour une course de chevaux

Quels sont les objets du système ainsi découverts ?

2. La centrale d'achat des chevaux

Construire le diagramme de séquence d'une prise de commande de produits sur le site *Internet* de la centrale d'achat des chevaux.

Quels sont les objets du système ainsi découverts ?

Introduction

L'objectif de ce chapitre 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 découvrons le diagramme des classes. Ce diagramme contient les attributs, les méthodes et les associations des objets. Comme nous l'avons vu au chapitre Les concepts de l'approche par objets, cette description est réalisée par les classes.

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 des classes pour exprimer de façon plus riche les contraintes. Enfin, le diagramme des objets nous montrera comment illustrer la modélisation réalisée dans le diagramme des classes.

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

Découvrir les objets du système par décomposition

Au chapitre La modélisation de la dynamique, nous avons étudié comment découvrir les objets d'un point de vue dynamique. Nous avons présenté les cas d'utilisation sous forme de diagrammes de séquence. Puis nous avons enrichi ces diagrammes au niveau de l'envoi de messages pour découvrir les objets du système.

La décomposition des messages fait apparaître les objets du système, car elle conduit à des messages plus fins dont il convient de rechercher le destinataire.

Une autre approche possible est la décomposition de l'information contenue dans un objet. Souvent, cette information est trop complexe pour n'être représentée que par la structure d'un seul objet. Elle doit parfois être également répartie entre plusieurs objets.

Exemple

Dans l'exemple du chapitre La modélisation de la dynamique, le directeur recherche les papiers (dans le sens d'informations) de la jument à vendre dans la base de données de l'élevage. Cette base constitue un objet à grosse granularité composé lui-même d'autres objets comme les papiers des chevaux, les informations financières et comptables, les documents d'achat et de vente de chevaux. Les papiers d'une jument sont composés, entre autres, de son carnet de vaccination et des papiers de ses descendants. Les papiers des descendants sont partagés par d'autres objets, comme les papiers de leur père éton. Cette décomposition est guidée par les données et non par des aspects dynamiques. La figure 6.1 illustre la composition de PapierJument dans le diagramme des classes.

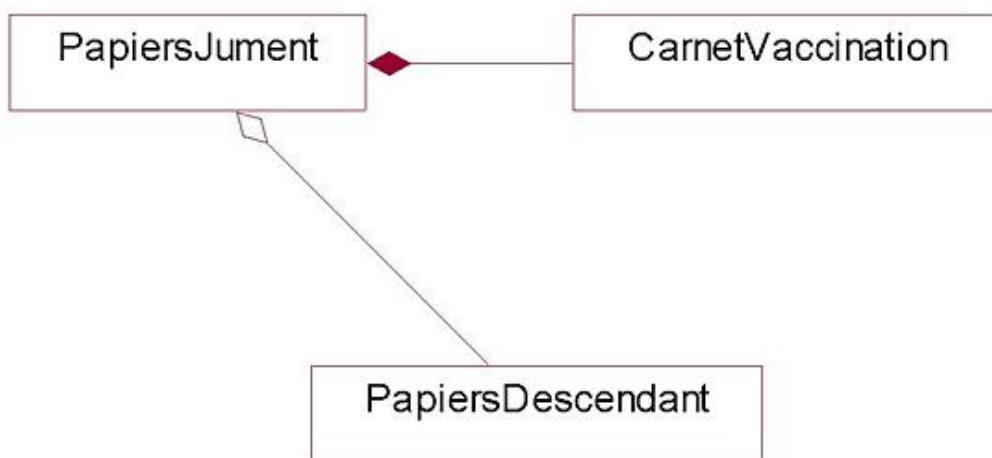


Figure 6.1 - Composition de PapiersJument

➤ Rappelons que la granularité d'un objet définit sa taille. Le système pris comme un objet est de gros grain ou de granularité importante. À l'opposé, le carnet de vaccination d'un cheval est un objet de grain beaucoup plus fin que le système.

Exemple

La décomposition d'un cheval pour faire apparaître ses différents organes peut se faire soit par la décomposition d'un diagramme de séquence, soit par la décomposition guidée par les données.

La décomposition par le diagramme de séquence consiste à analyser différents envois de message : faire peur, courir, manger, dormir. Ces derniers feront apparaître progressivement les différents organes du cheval. Elle est illustrée à la figure 6.2 pour le message fairePeur. Un cheval dilate ses naseaux pour marquer l'alerte, la surprise ou la peur. Il pince la bouche pour indiquer tension, peur ou colère. Enfin, la ruade est un mouvement défensif.

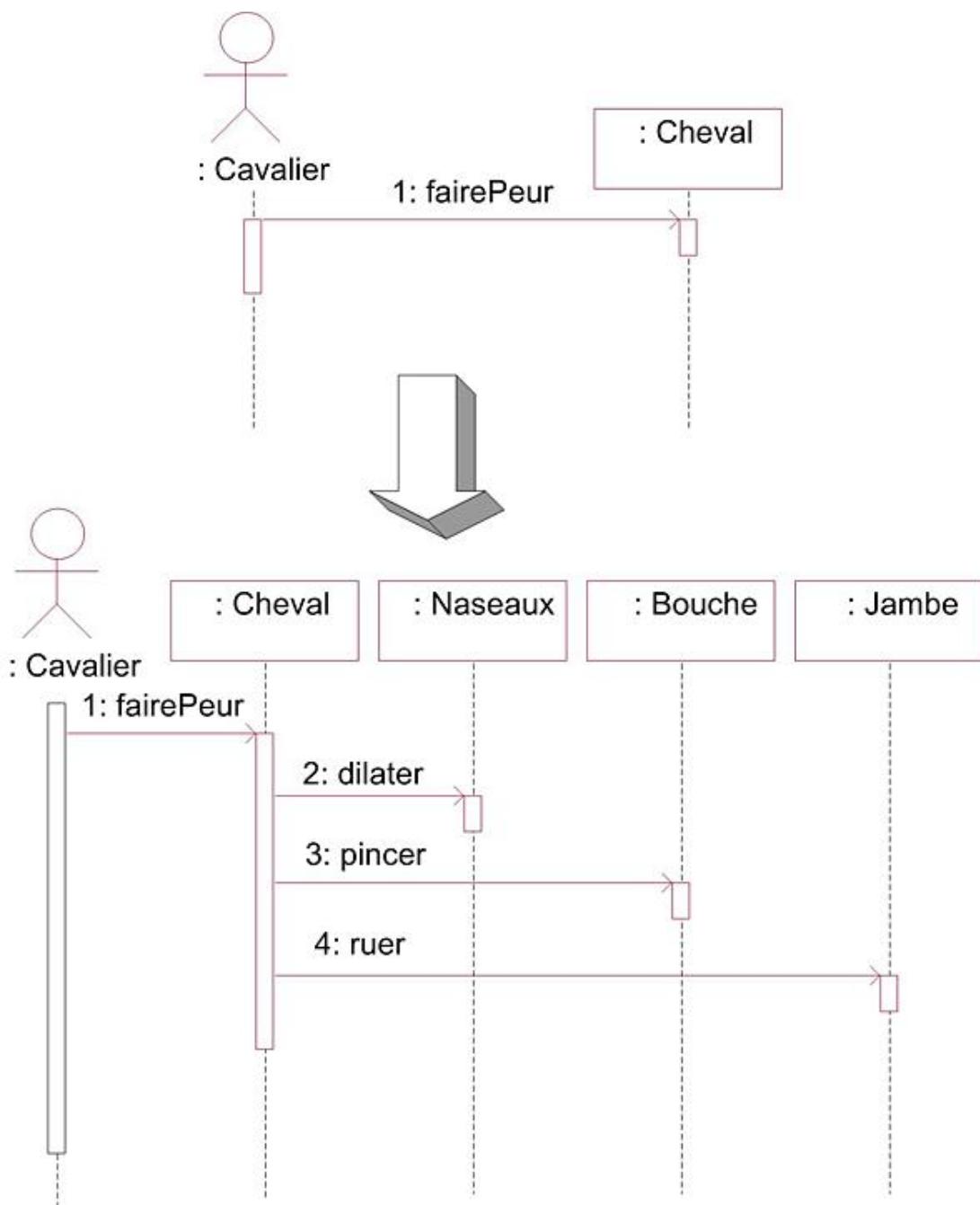


Figure 6.2 - Découverte des objets par enrichissement du diagramme de séquence

La décomposition guidée par les données consiste à étudier directement les différents organes d'un cheval et à les prendre en compte dans le diagramme des classes. La figure 6.3 illustre un cheval composé de ses différents organes.

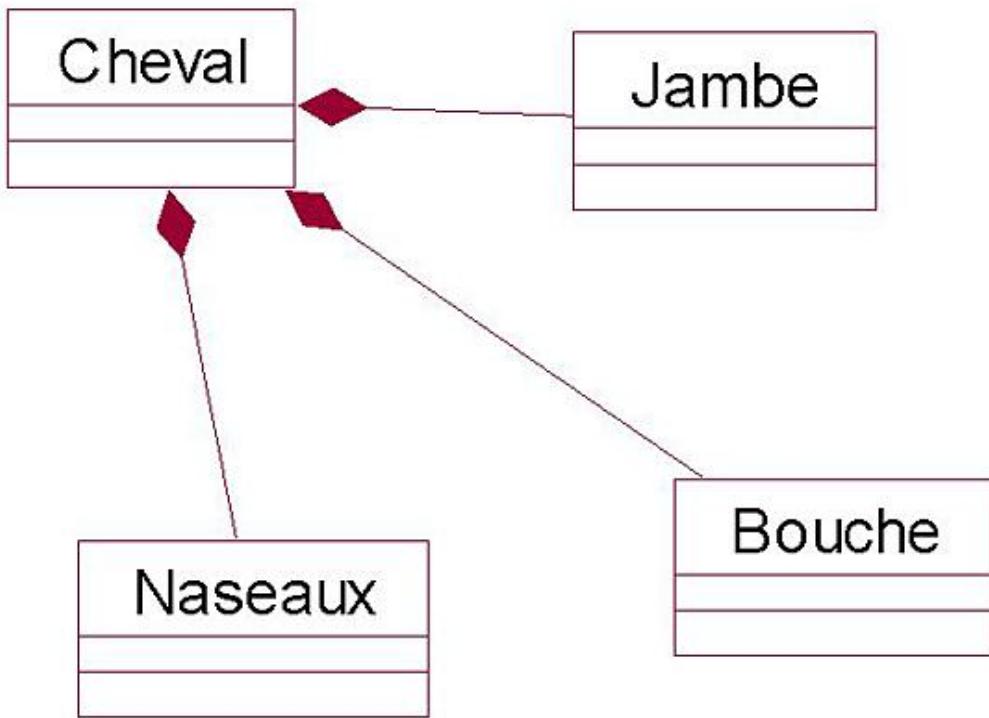


Figure 6.3 - Composition de Cheval

-
- La décomposition par le diagramme de séquence, comme la décomposition guidée par les données apparaissent naturelles pour découvrir les objets. Ce résultat est normal car un objet est l'assemblage d'une structure et d'un comportement. Il convient enfin de remarquer que ces deux approches ne sont pas incompatibles.

 - La décomposition guidée par les données est plus efficace quand la personne chargée de la modélisation connaît bien le domaine. La décomposition en objets est alors immédiate.

La représentation des classes

1. La forme simplifiée de 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 6.4. Cette représentation est constituée de trois parties.

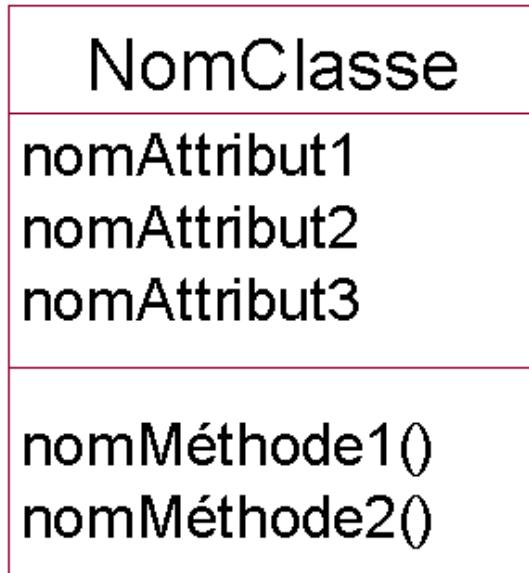


Figure 6.4 - Représentation simplifiée d'une classe en UML

La première partie contient le nom de la classe.

➤ Rappelons que le nom d'une classe est au singulier. Il est toujours 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.

➤ 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



Figure 6.5 - La classe Cheval

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.

2. L'encapsulation

L'encapsulation a été introduite dans le chapitre Les concepts de l'approche par objets. 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 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 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 paquetage sera abordée au chapitre La structuration des éléments de modélisation.

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.

➤ La plupart des modélisations emploient cependant l'encapsulation privée mais, à notre avis, sans que leur auteur se soit soucié de la différence existant entre encapsulation privée, protégée ou de paquetage. C'est ce que nous avons fait dans le chapitre Les concepts de l'approche par objets, par souci de simplification.

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 6.6 montre la classe Cheval en faisant ressortir les caractéristiques d'encapsulation.

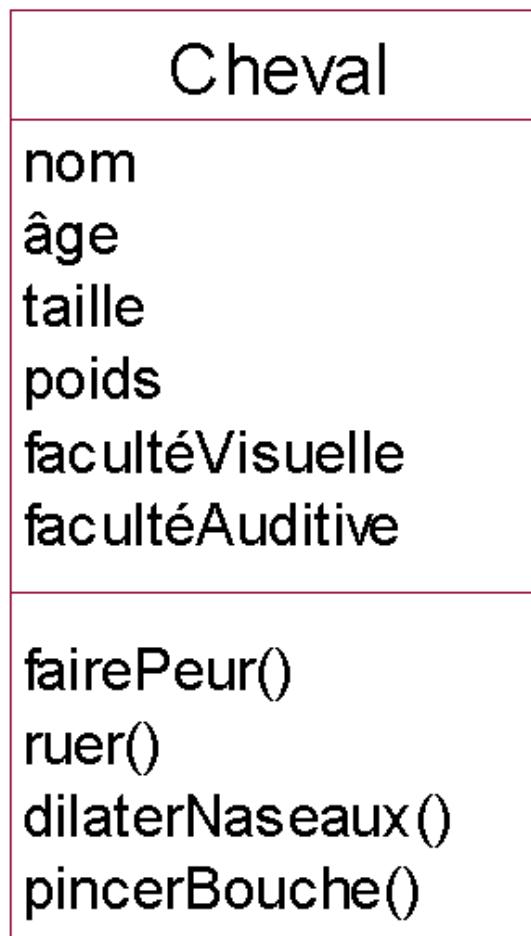


Figure 6.6 - La classe Cheval avec les caractéristiques d'encapsulation

3. La notion de type

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 standard 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 le type d'un attribut ne fait généralement recours à une classe que si cette dernière est une classe d'une bibliothèque externe au système modélisé ou une interface comme nous les étudierons à la fin du chapitre. Il n'est pas conseillé d'utiliser une classe du système pour donner un type à un attribut. Dans ce cas, il vaut mieux recourir aux associations interobjets, comme nous l'étudierons dans la suite de ce chapitre.

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 6.7 montre la classe Cheval pour laquelle le type de tous les attributs a été fixé. Les attributs de cette classe utilisent des types standard.

<h1>Cheval</h1>
<p>+ nom : String + âge : Integer + taille : Integer + poids : Integer # facultéVisuelle : Integer # facultéAuditive : Integer</p>
<p>+ fairePeur() # ruer() # dilaterNaseaux() # pincerBouche()</p>

Figure 6.7 - La classe Cheval avec attributs typés

4. 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.

Comme nous l'avons vu précédemment, ces paramètres ainsi que le résultat peuvent être typés. L'ensemble constitué du nom de la méthode, des paramètres avec leur nom et leur type ainsi que le type du résultat s'appelle la signature de la méthode. Une signature prend la forme suivante :

```
<nomMéthode> (<direction><nomParamètre> : <type>, ...) :
<typeRésultat>
```

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.

 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 6.8 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.

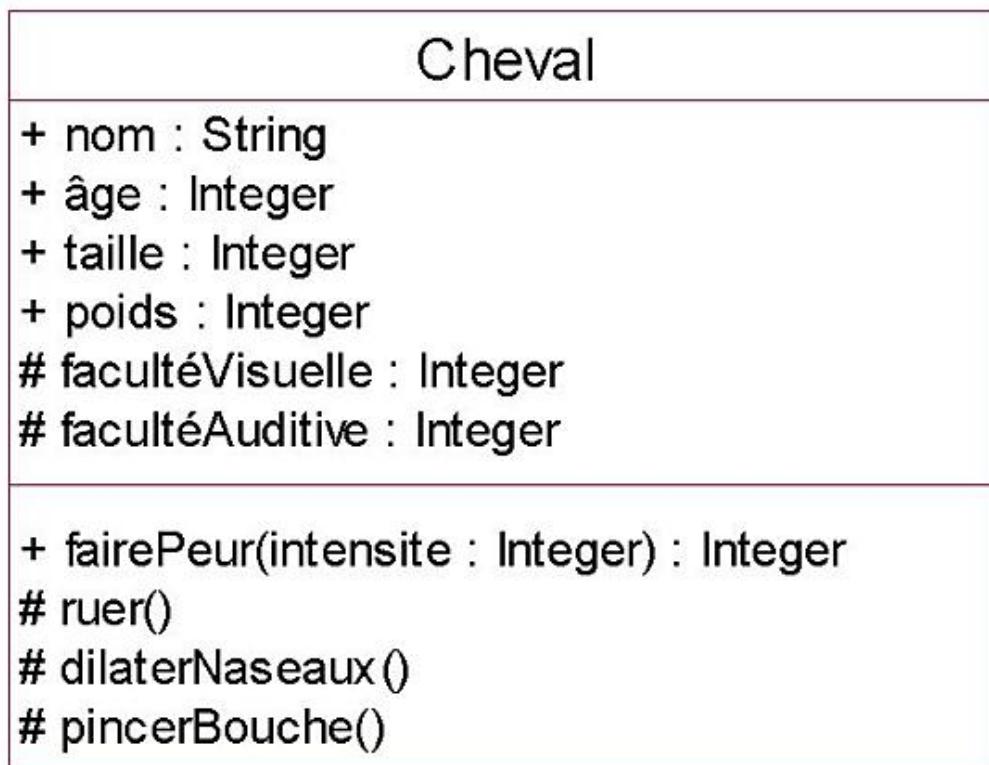


Figure 6.8 - La classe Cheval avec la signature des méthodes

5. La forme complète de représentation des classes

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 donner 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 6.9 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.

NomClasse
+ nomAttribut1 : typeAttribut1 = valeurDéfaut
nomAttribut2 : typeAttribut2 = valeurDéfaut
+ nomAttribut3 : typeAttribut3 = valeurDéfaut
+ nomMéthode1(param1 : typeParam1 = valeurDéfaut, param2 : typeParam2) : typeRetour
nomMéthode2(param : typeParam = valeurDéfaut) : typeRetour

Figure 6.9 - Représentation complète d'une classe en UML

➤ 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.

6. 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 6.10 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.

QuartierViandeChevaline
+ poids : Integer
+ qualité : Integer
prixSansTVA : Currency
<u># tauxTVA : Integer = 55</u>
+ prixAvecTVA() : Currency

Figure 6.10 - Attribut de classe

➤ 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. Une telle méthode ne manipule que les attributs de classe.

Exemple

La figure 6.11 ajoute une méthode de classe à la classe QuartierViande-Chevaline qui sert à fixer le taux de TVA. En effet, celui-ci est fixé par la loi et cette dernière peut être modifiée.

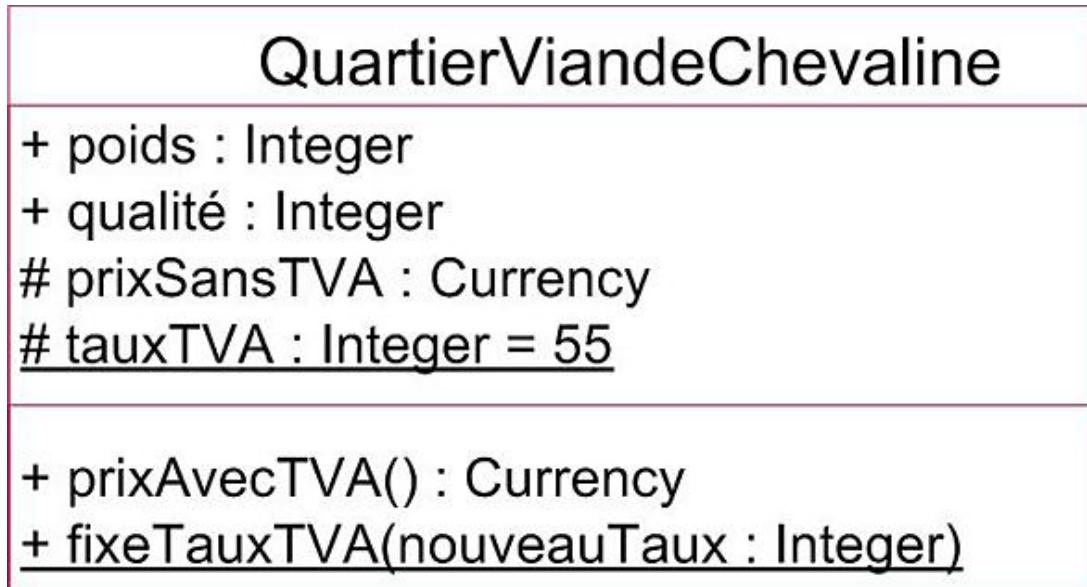


Figure 6.11 - Méthode de classe

➤ Dans de nombreux outils UML, les termes "attribut et méthode de classe" ne sont pas utilisés. Ces outils préfèrent les termes de "attributs ou méthodes statiques". Ces termes sont employés dans les langages de programmation modernes 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 sous-classes. Une sous-classe peut accéder à un attribut ou à une méthode de classe de l'une de ses sous-classes 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.

➤ Rappelons aussi qu'une sous-classe peut accéder à un attribut ou à une méthode de classe de l'une de ses sous-classes, à la condition que l'encapsulation privée n'ait pas été utilisée.

7. Les attributs calculés

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 contrainte donnant le moyen de calculer sa valeur.

Exemple

Nous reprenons l'exemple de la figure 6.10. La méthode `prixAvecTVA` est remplacée par l'attribut calculé `/prixAvecTVA` comme l'illustre la figure 6.12.

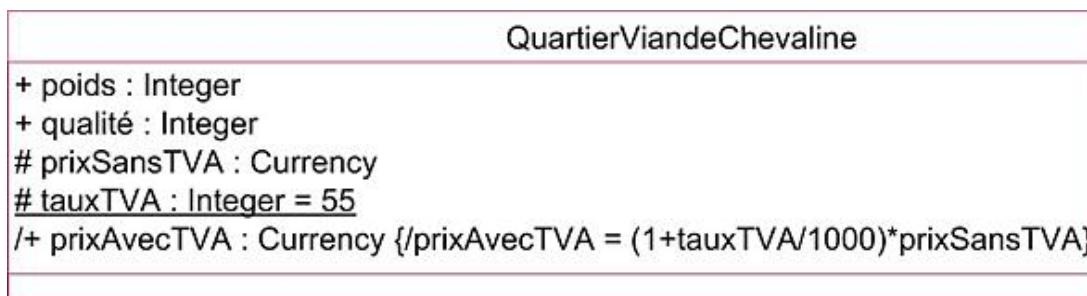


Figure 6.12 - Attribut calculé

- Toute contrainte exprimée dans un diagramme UML est écrite entre accolades.
-

Les associations entre objets

1. Les liens 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.*

➤ 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.

2. La représentation des associations entre les classes

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

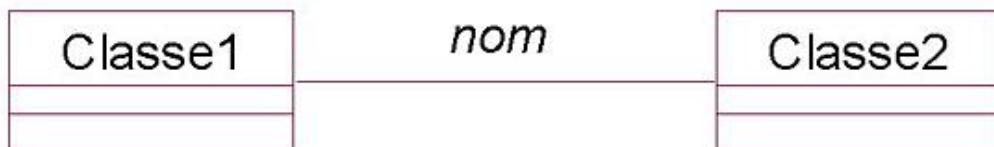


Figure 6.13 - Association binaire entre classes

➤ 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 pour 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 6.14 illustre la représentation d'une association binaire en montrant les rôles.

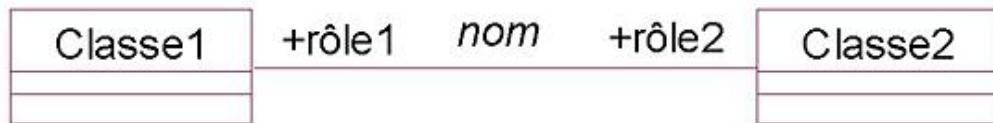


Figure 6.14 - Rôles d'une association binaire

Exemple

La figure 6.15 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.

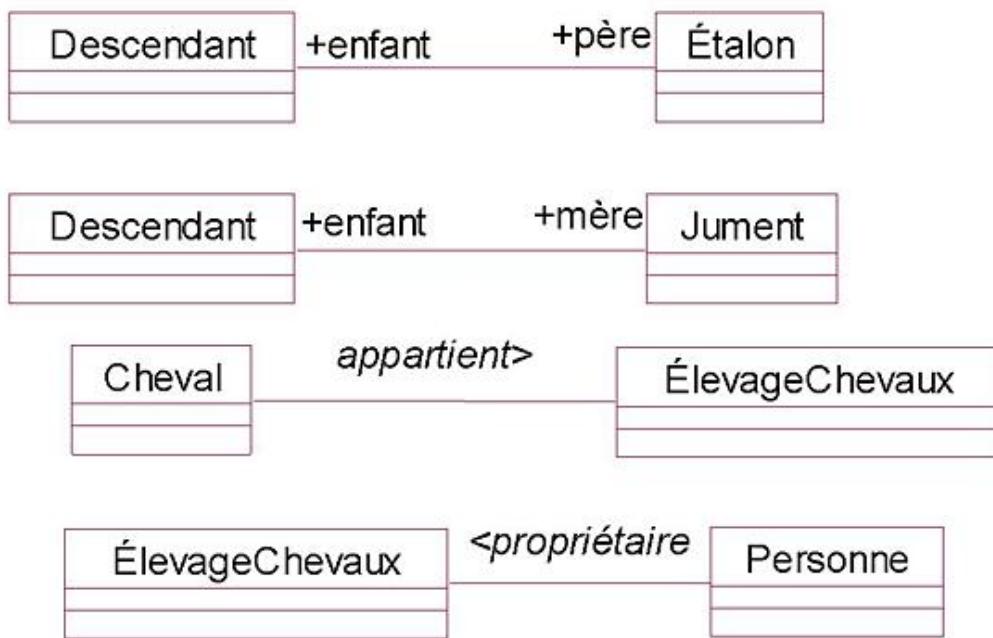


Figure 6.15 - Exemples d'associations binaires

La représentation graphique d'une association ternaire et au-delà consiste en un losange qui relie les différentes classes. La figure 6.16 illustre la représentation d'une association ternaire.

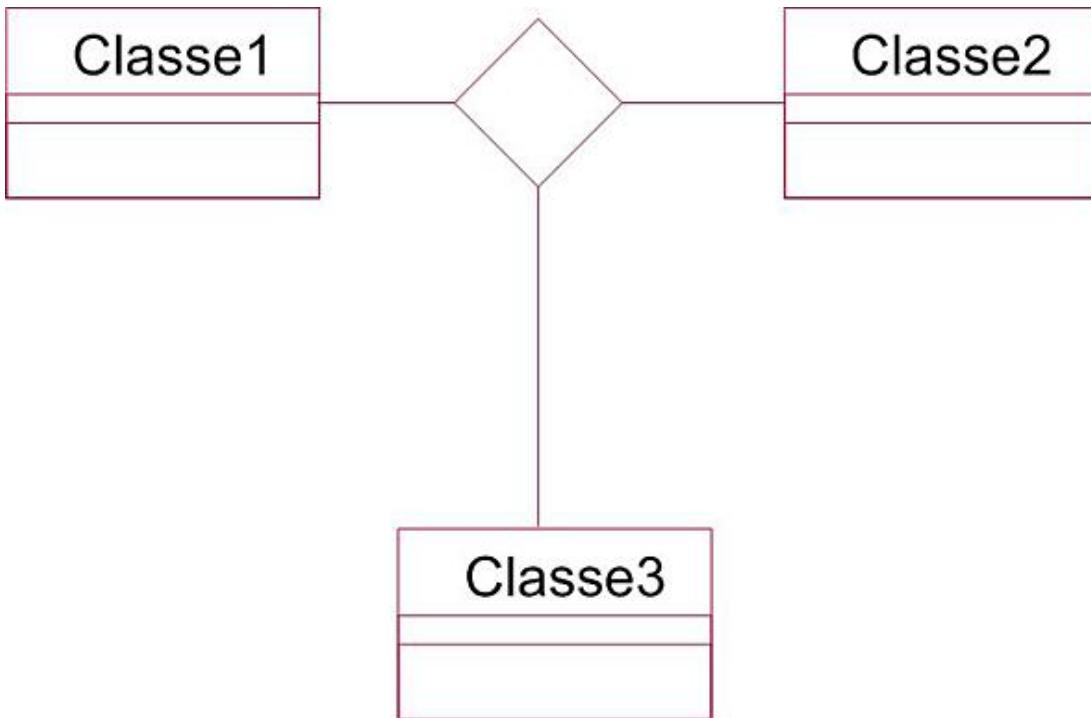


Figure 6.16 - Association ternaire entre classes

Exemple

L'association famille qui relie les classes *Étalon*, *Jument* et *Descendant* est illustrée à la figure 6.17. Chacune de ses occurrences constitue un triplet (*père*, *mère*, *poulain*).

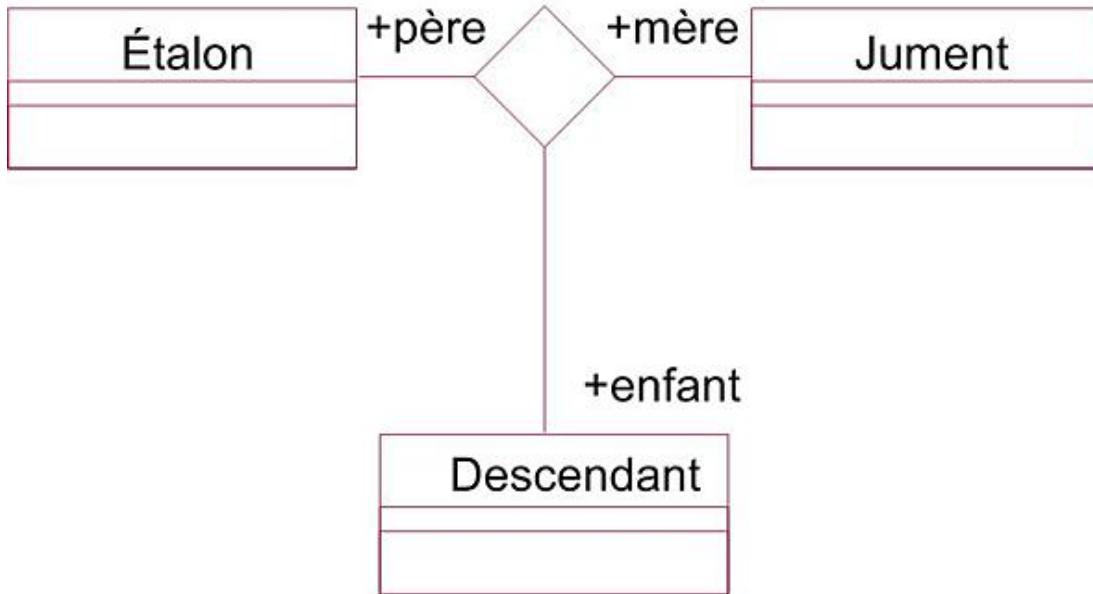


Figure 6.17 - Exemples d'association ternaire

3. La cardinalité des associations

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 minimales et maximales 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 un à plusieurs fois
M..N	entre M et N fois
N	N fois

➤ En l'absence de spécification explicite des cardinalités minimales et maximales, celles-ci valent 1.

Exemple

En figure 6.18, nous reprenons nos exemples et y ajoutons les cardinalités minimales et maximales de chaque association. Un élevage peut avoir plusieurs copropriétaires et une personne peut être propriétaire de plusieurs élevages.

Pour illustrer la lecture, nous indiquons comment lire la première association : Un descendant possède un seul père. Un étalon peut avoir de zéro à plusieurs enfants.

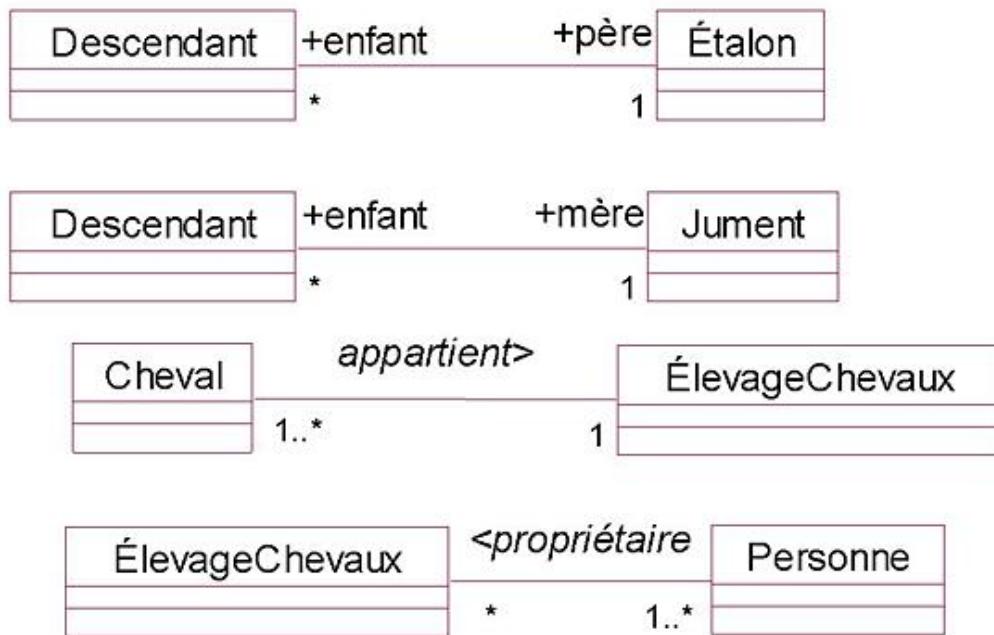


Figure 6.18 - Exemples d'associations décrites avec leurs cardinalités

4. Navigation

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 proche 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 n'est pas nécessaire.

La figure 6.19 illustre l'association résultant avec le sens de navigation adapté.

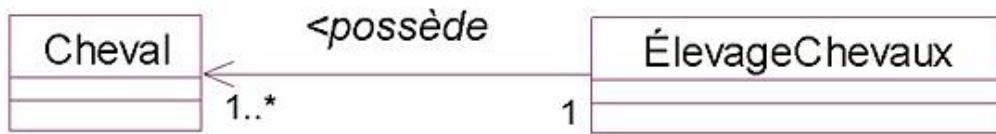


Figure 6.19 - Exemple de navigation

5. Associer une classe avec elle-même

La même classe peut se trouver 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.

 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 6.20 illustre cette association. Cette association crée une hiérarchie au sein des concours.

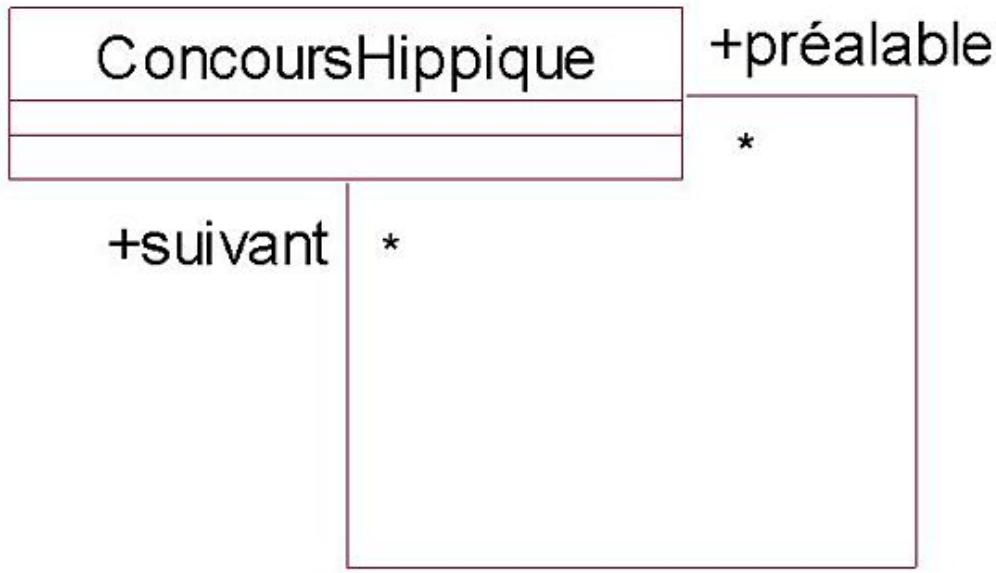


Figure 6.20 - Association réflexive entre concours hippiques

Exemple

La figure 6.21 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.

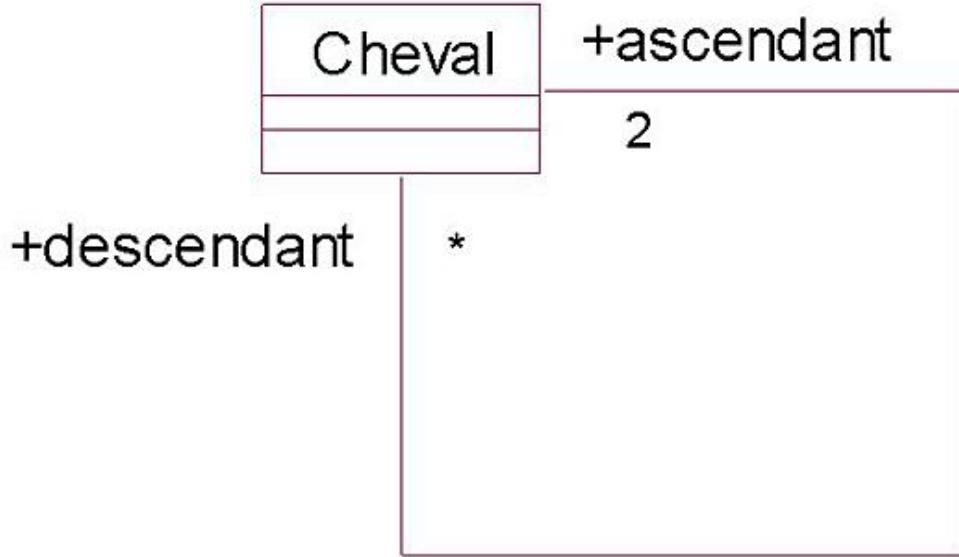


Figure 6.21 - Association "ascendant/descendant direct" entre chevaux

Exemple

La figure 6.22 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.

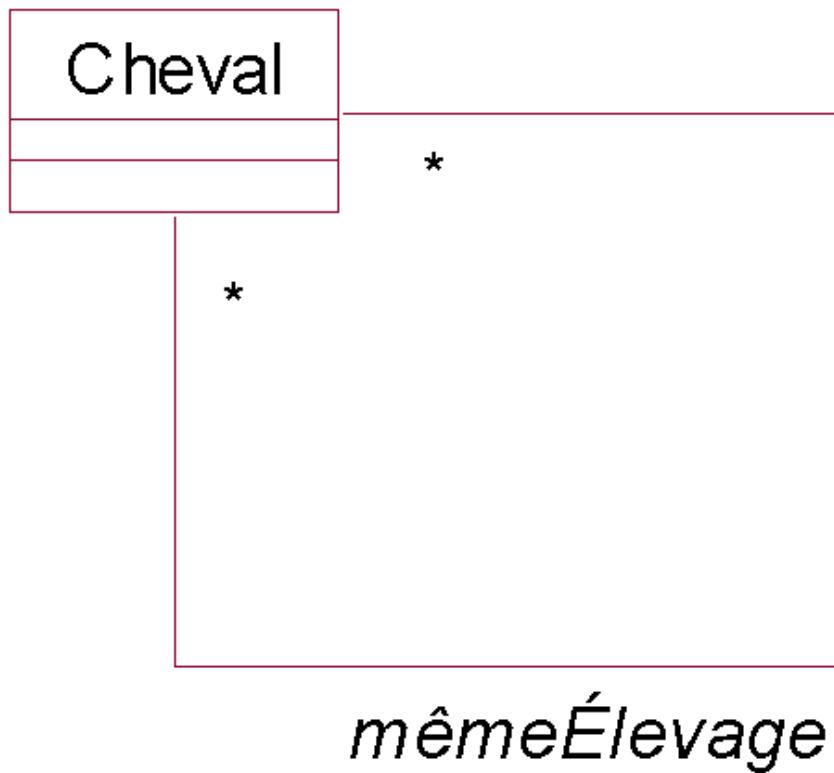


Figure 6.22 - Association réflexive qui relie les chevaux d'un même élevage

6. Les classes-associations

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

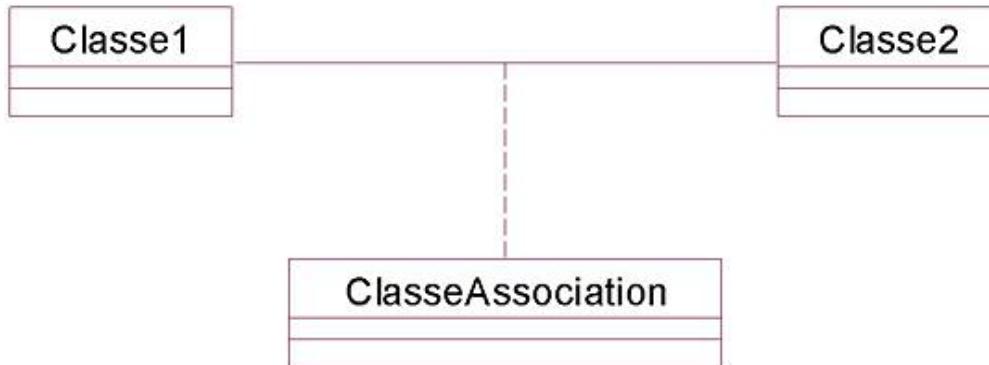


Figure 6.23 - Représentation graphique d'une classe-association

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 (voir figure 6.24).

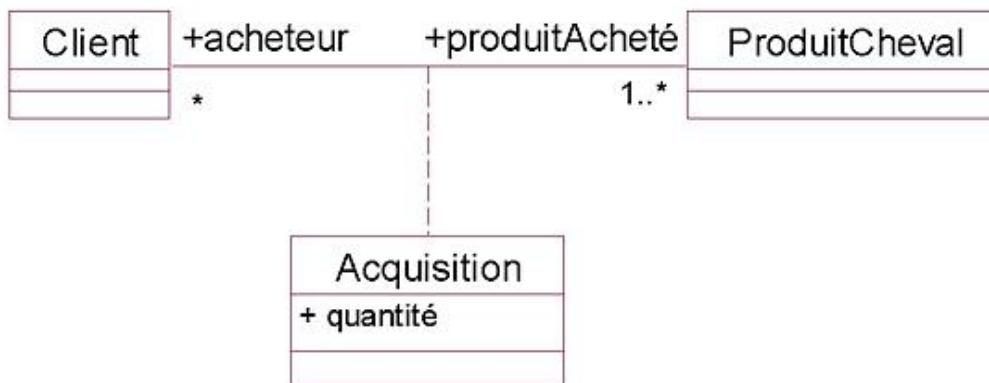


Figure 6.24 - Classe-association Acquisition

7. La qualification des associations

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 clef, par exemple pour retrouver une ligne dans une base de données relationnelle.

Cette qualification est alors insérée à l'autre extrémité sous la forme d'un ou de plusieurs attributs (voir figure 6.25) où les instances de la classe 2 sont qualifiées au niveau de la classe 1.

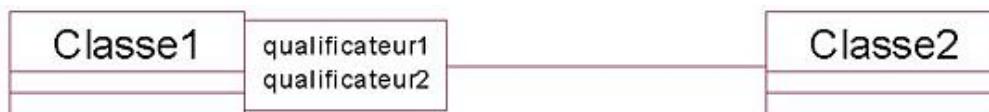


Figure 6.25 - Représentation graphique d'une qualification

Exemple

Un tableau est constitué d'éléments. La figure 6.26 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 un.

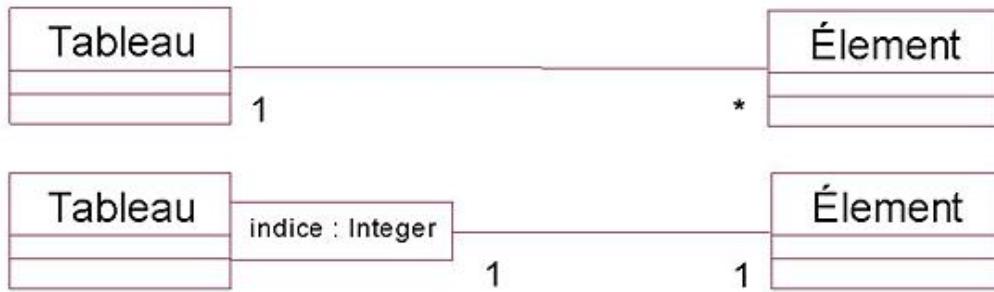


Figure 6.26 - Qualification des éléments d'un tableau

Exemple

Lors d'une course de chevaux, chaque cheval est numéroté. La figure 6.27 illustre l'ensemble des participants d'une course en les qualifiant par leur numéro.

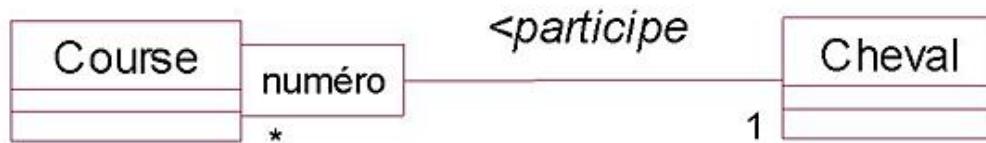


Figure 6.27 - Qualification des participants à une course hippique

8. L'expression de contraintes sur les associations

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.

Cependant, ces types de contraintes peuvent se révéler insuffisantes. Pour d'autres contraintes, UML propose de les exprimer en langage naturel.

Il existe également le langage OCL, langage de contraintes *objet* sous forme de conditions logiques. Ce langage OCL fait partie de l'ensemble de la notation UML.

Qu'elles soient écrites en langage naturel ou en OCL, ces deux types de contraintes sont représentés 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 (collections d'objets), OCL propose un jeu d'opérateurs : includes, intersection, union, forEach, exists, etc.

Pour désigner un attribut ou une méthode dans une expression OCL, il faut élaborer une expression de chemin qui commence par le nom de la classe. 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.

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

Classe.role.role.role.attribut

ou

Classe.role.role.role.méthode

 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 repas pour un cheval contient les éléments suivants :

- paille ;
- minéraux ;
- foin ou granulés.

La figure 6.28 illustre le repas et ses différents constituants. Qu'il contienne soit du foin, soit des granulés est une contrainte exprimée par l'opérateur ou. Celui-ci exprime l'exclusion entre les deux associations.

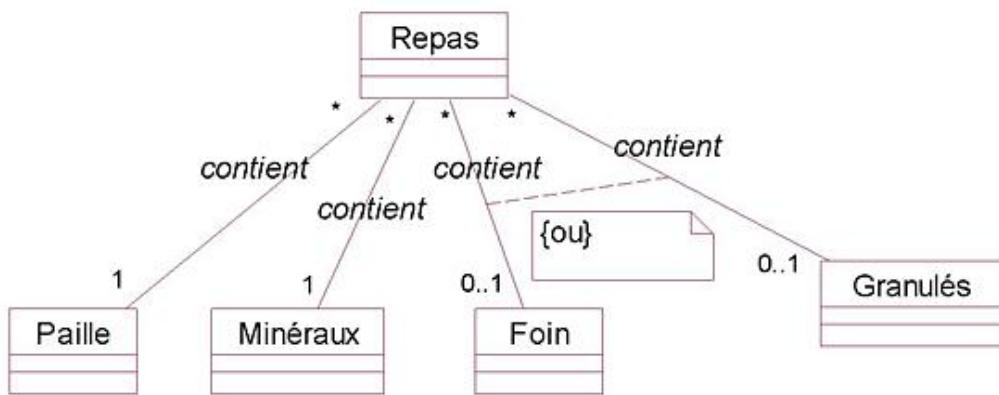


Figure 6.28 - Expression de l'exclusion entre deux associations

Exemple

Un hippodrome fait courir certains chevaux. Il doit alors recruter les jockeys choisis par les propriétaires de ces chevaux pour les monter. Ceci peut être reformulé ainsi : L'ensemble de jockeys recrutés 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 :

```
Hippodrome.participeCourse.monte  
->includesAll(Hippodrome.employé)
```

Une telle contrainte s'appuie sur l'utilisation des expressions de chemin en OCL. La figure 6.29 illustre le diagramme de classes où la contrainte écrite en OCL est incluse dans une note.

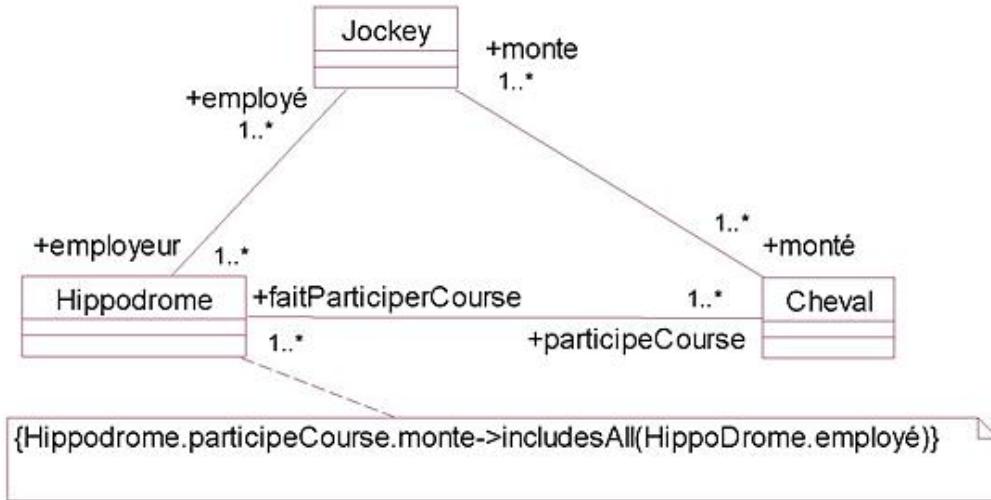


Figure 6.29 - Contrainte avec expressions de chemin en OCL

9. Les objets composés

Au chapitre Les concepts de l'approche par objets, nous avons vu qu'un objet peut être composé d'autres objets. Dans ce cas, il s'agit d'une association entre objets d'un cas particulier appelé *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, que nous allons examiner maintenant.

a. 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 de un.

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

La figure 6.30 présente la représentation graphique de l'association de composition forte. Au niveau de l'objet composé, la cardinalité minimale a été indiquée à zéro mais elle pourrait aussi être de un.



Figure 6.30 - Association de composition forte

Par la suite, l'association de composition forte sera appellée 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 6.31.



Figure 6.31 - Association de composition entre un cheval et son cerveau

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 6.32.

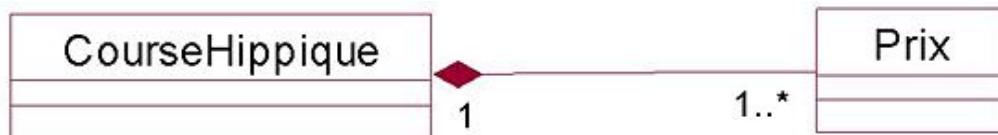


Figure 6.32 - Association de composition entre une course hippique et les prix dont elle est constituée

b. La composition faible ou agrégation

La composition faible, appelée plus couramment agrégation, impose beaucoup moins de contraintes aux composants que la composition forte étudiée jusqu'à présent. 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.

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 également les contraintes explicites 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 requiert 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 6.33). 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.

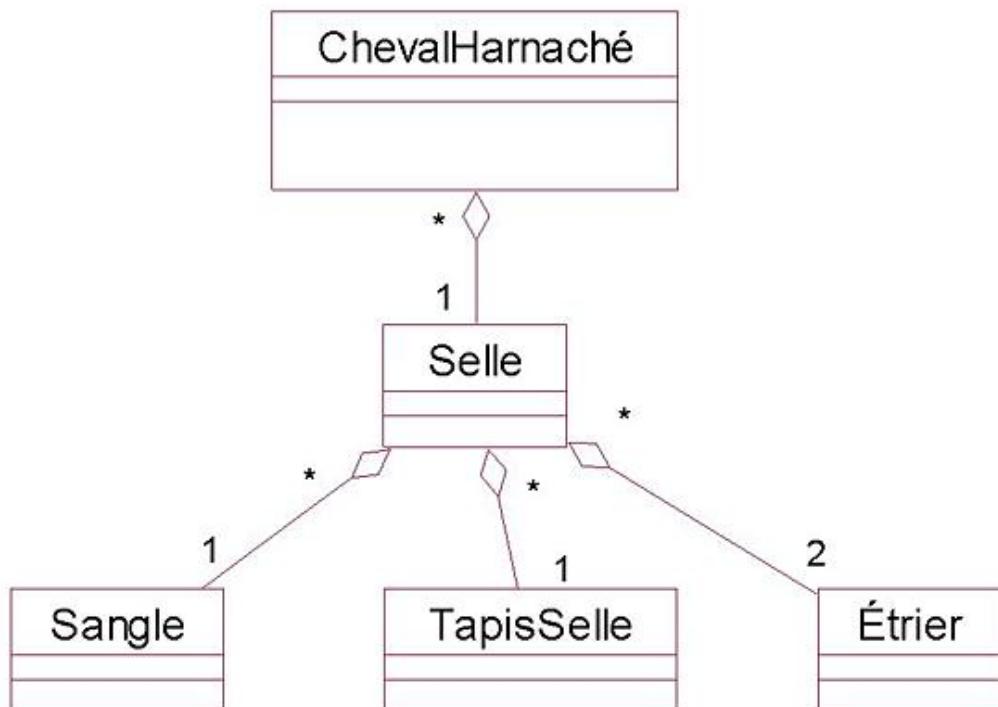


Figure 6.33 - Association d'agrégation entre un cheval harnaché et son équipement et entre une selle et ses composants

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 6.34.

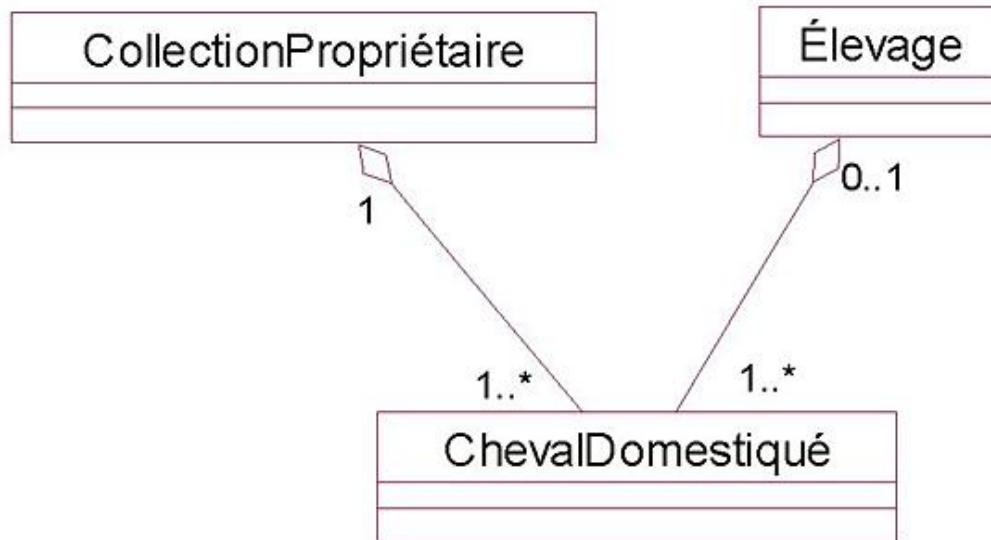


Figure 6.34 - Composant partagé par plusieurs agrégations distinctes

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 6.35).

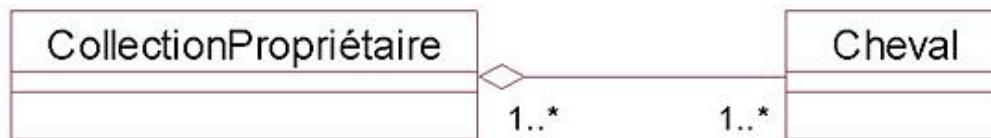


Figure 6.35 - Composant partagé plusieurs fois dans une même agrégation

c. 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 généralisation/specialisation entre les classes

1. Classes plus spécifiques et 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 6.36 illustre ces deux classes ainsi que la relation de généralisation/specialisation qui les relie.

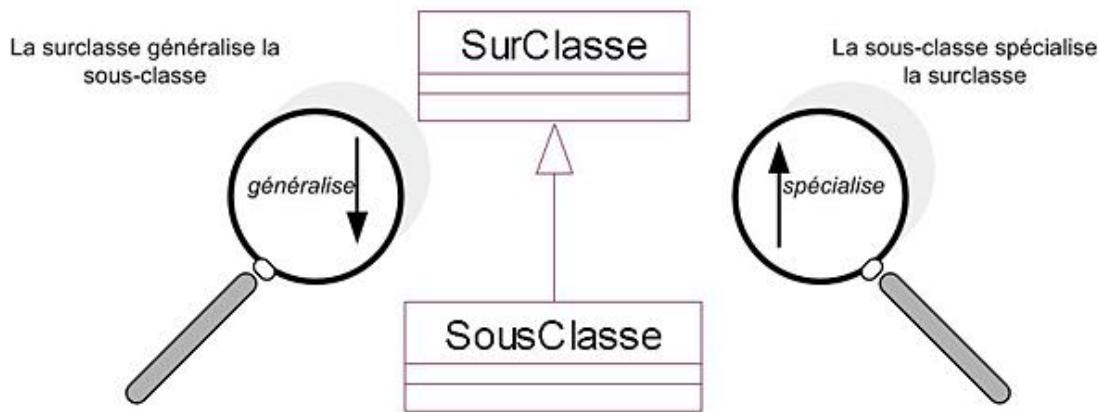


Figure 6.36 - Sous-classe et surclasse

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 6.37.

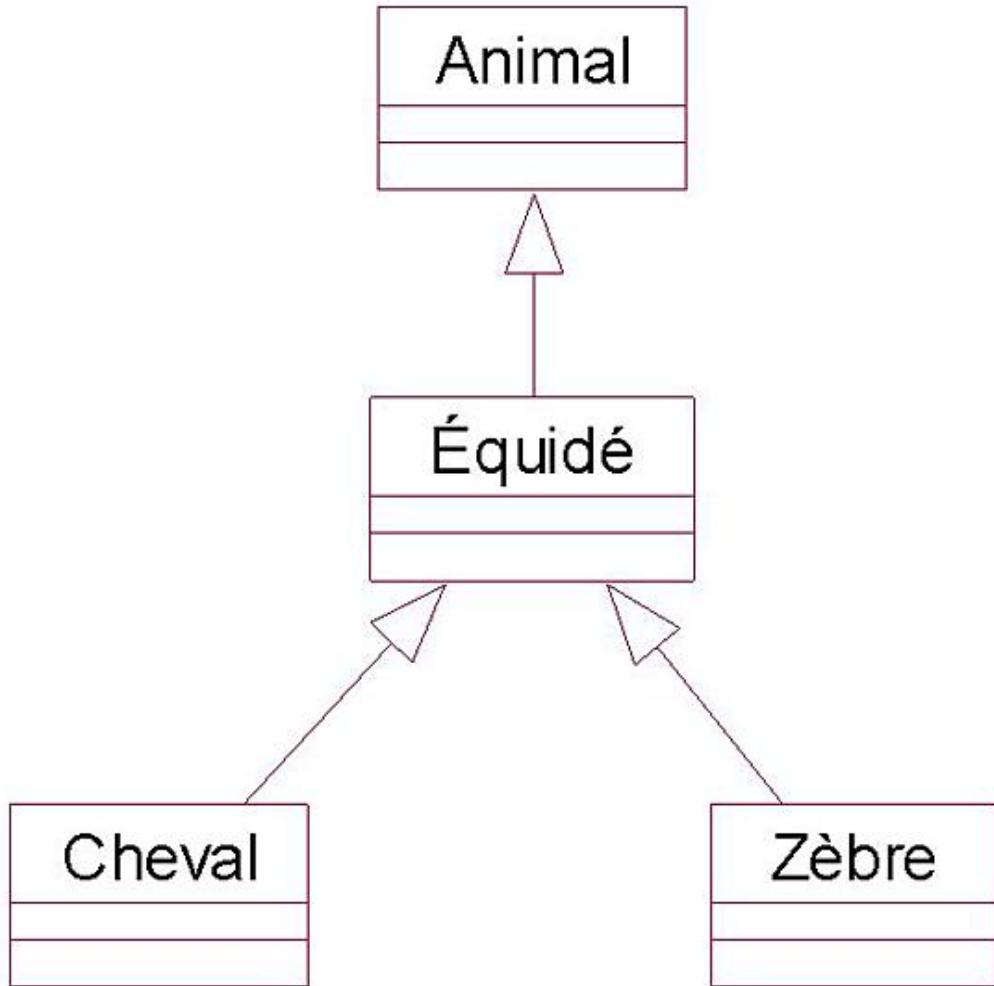


Figure 6.37 - Hiérarchie de classes

2. 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.

➤ 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 peut être redéfinie dans la sous-classe. Nous verrons par la suite que 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 telles que l'illustre la figure 6.38.

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.

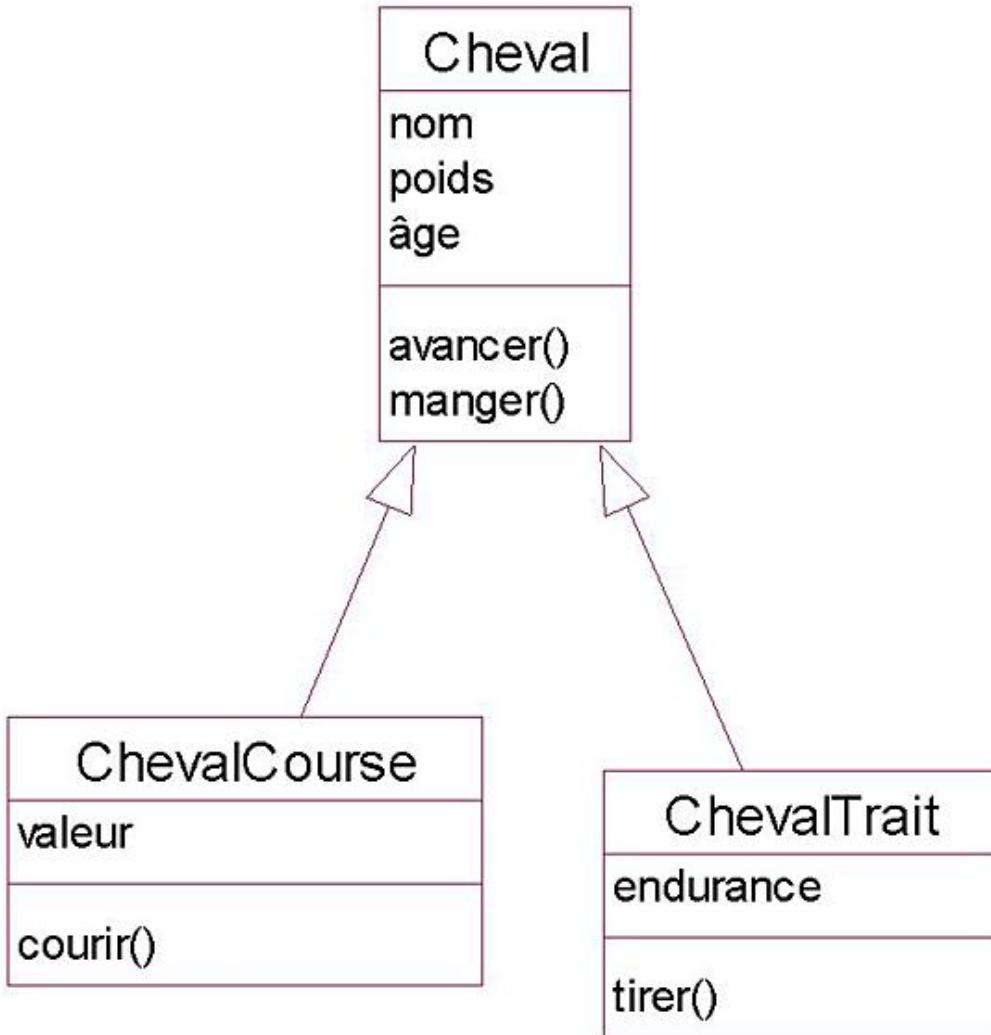


Figure 6.38 - Héritage d'attributs et de méthodes

3. Classes concrètes et abstraites

La figure 6.37 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 et les classes abstraites *Mammifère* et *Animal*.

Une classe concrète possède des instances. Elle constitue un modèle complet d'objet (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 italiques 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 6.39 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 que le cheval est un herbivore.

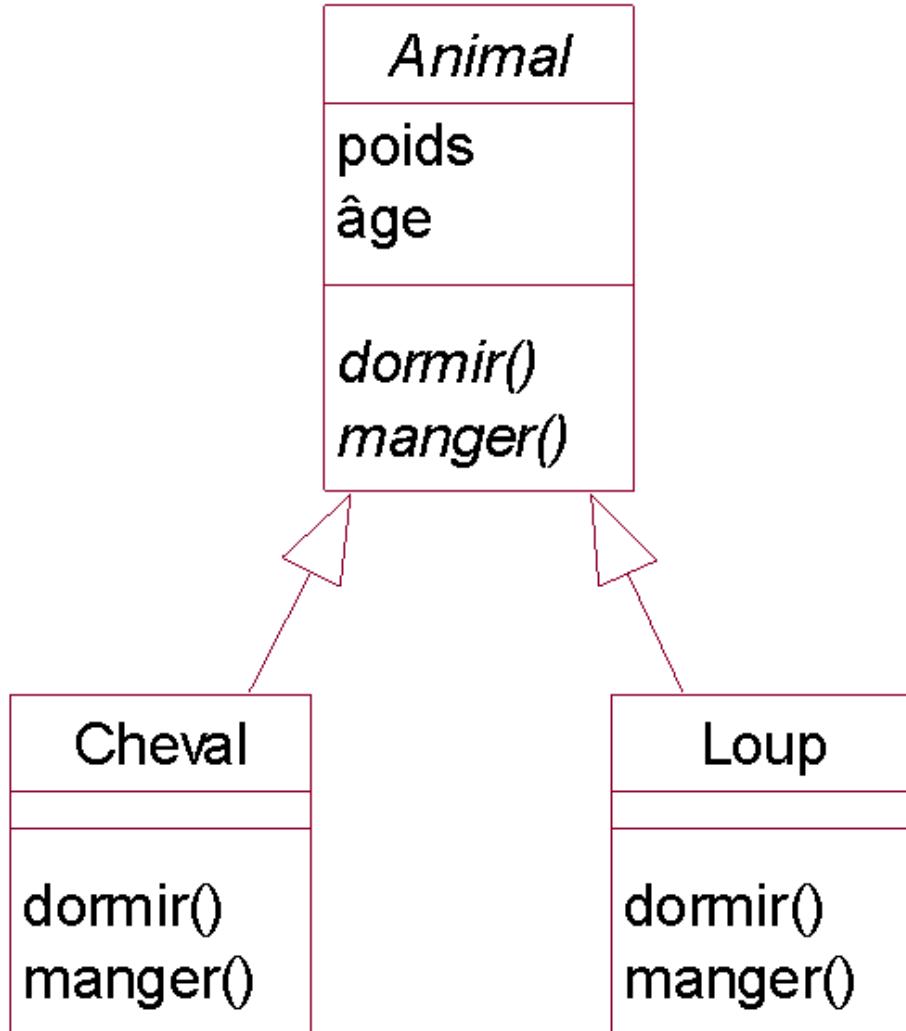


Figure 6.39 - Classes et méthodes abstraites

➤ Rappelons que la signature est composée de l'ensemble constitué du nom de la méthode, des paramètres avec leur nom et leur type ainsi que le 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 ne soit pas une description complète d'objets.

4. Expression de 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 6.40 illustre une relation d'héritage entre la surclasse Équidé et deux sous-classes : les chevaux et les ânes. 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. De ce fait, la figure fait ressortir les contraintes {incomplete} et {overlapping}.

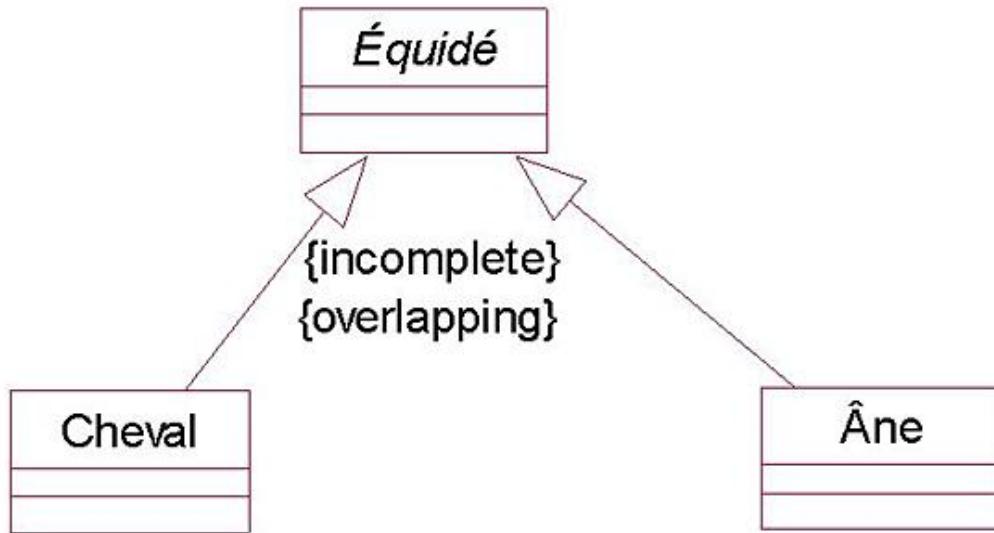


Figure 6.40 - Sous-classes sans couverture et avec instances communes

Exemple

La figure 6.41 illustre une autre relation d'héritage entre la surclasse Cheval et deux sous-classes : les chevaux mâles et les chevaux femelles. Ces deux sous-classes couvrent la classe des chevaux et il n'existe aucun cheval qui soit à la fois mâle et femelle. De ce fait, la figure fait ressortir les contraintes {complete} et {disjoint}.

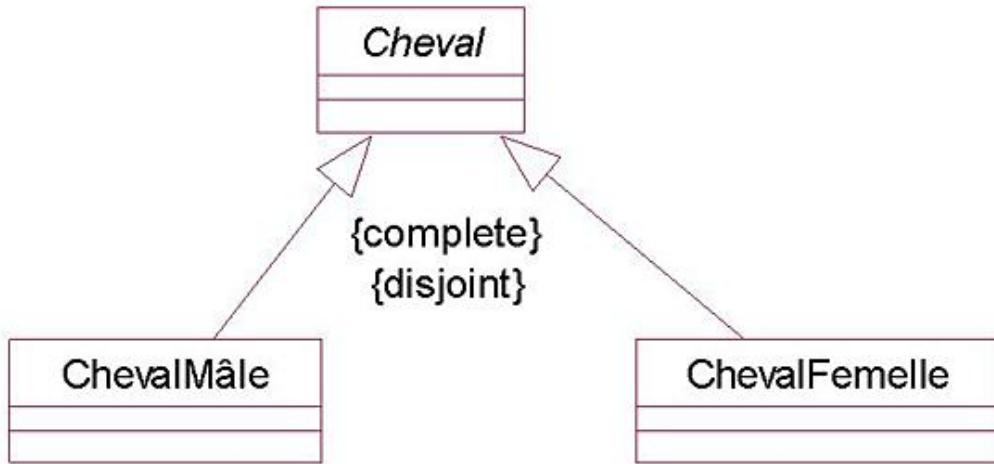


Figure 6.41 - Sous-classes avec couverture et sans instance commune

5. L'héritage multiple

L'héritage multiple en UML consiste à ce qu'une sous-classe hérite de plusieurs surclasses. Il pose un seul problème : il engendre un conflit lorsqu'une même méthode, c'est-à-dire de 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.

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 6.42 reprend les deux sous-classes Cheval et Âne et introduit une sous-classe commune à savoir les mulets.

La figure 6.42 illustre également la méthode trotter, abstraite dans la classe Équidé, 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.

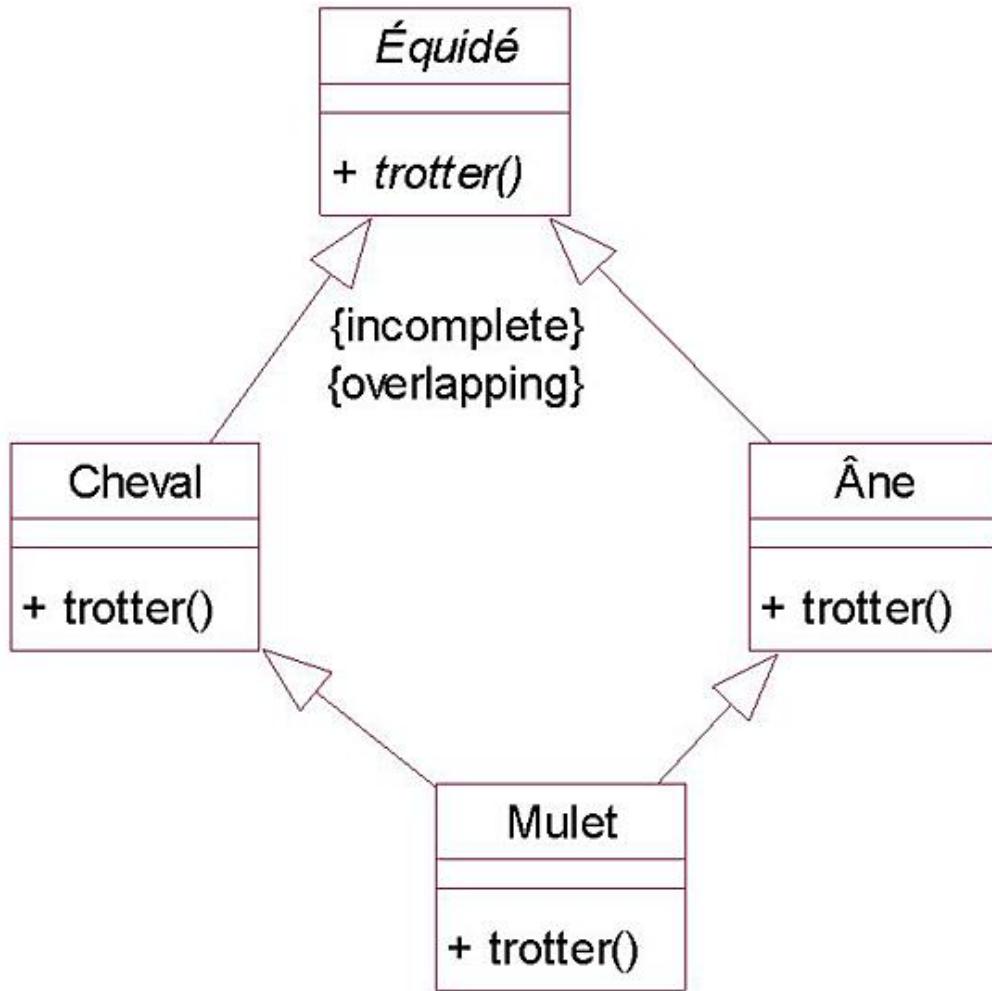


Figure 6.42 - Héritage multiple avec conflit

6. Factorisation des relations entre objets

Nous avons vu qu'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 figure 6.43).

La figure 6.44 montre qu'une fois créée, la classe abstraite Mammifère surclasse des classes Cheval et Loup, les deux associations de composition sont factorisées.

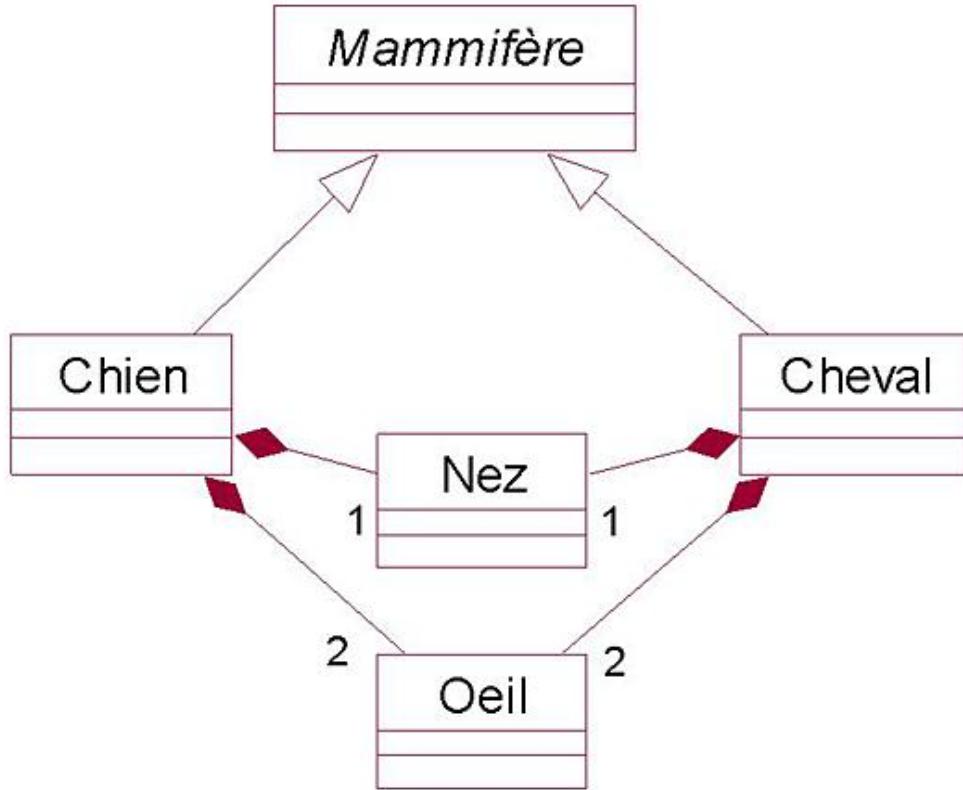


Figure 6.43 - Associations entre objets non factorisées

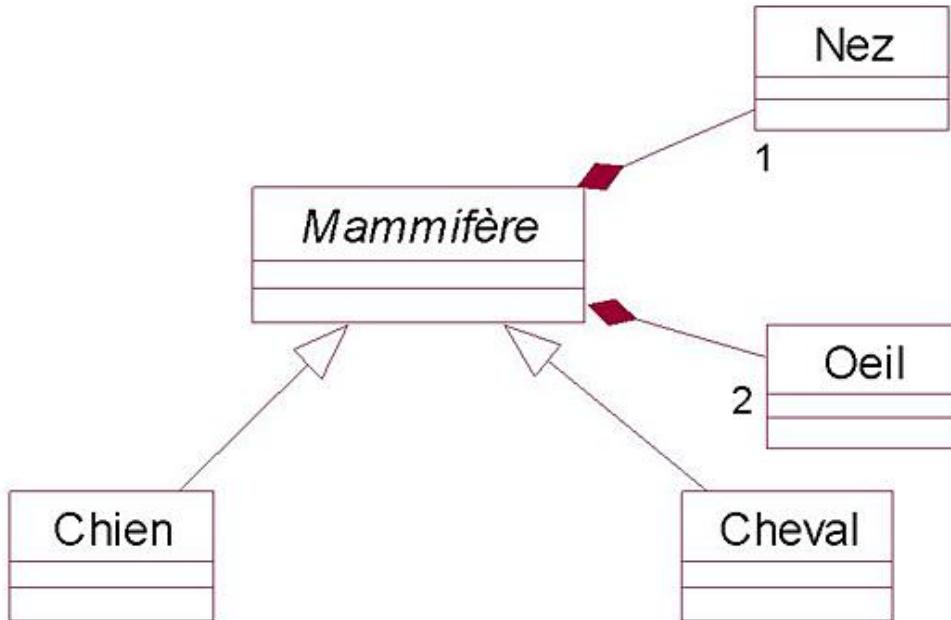


Figure 6.44 - Associations entre objets factorisées

7. Interface

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 6.45 illustre cette représentation.

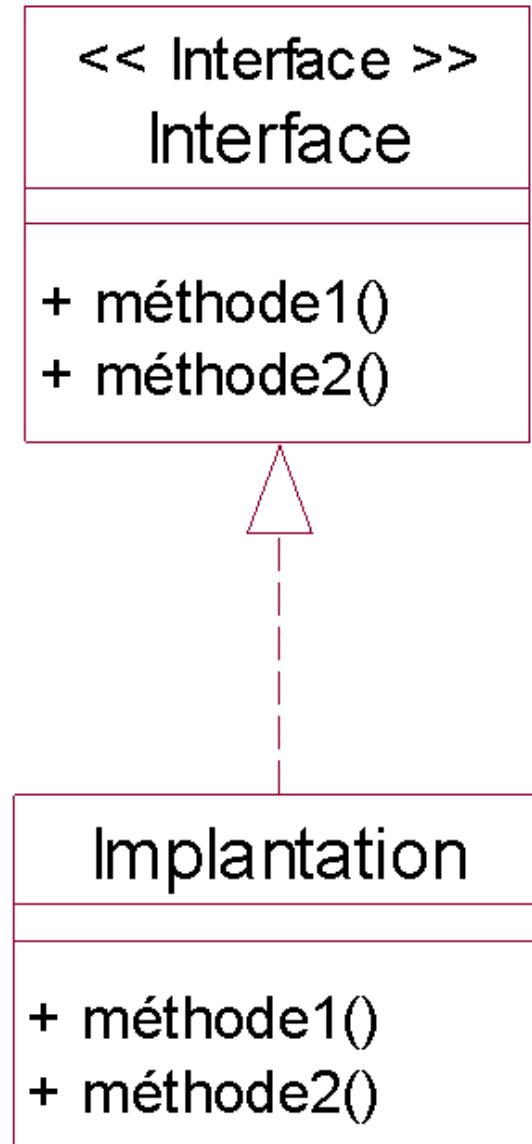


Figure 6.45 - Interface et relation de réalisation

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.

La figure 6.46 illustre l'exemple.

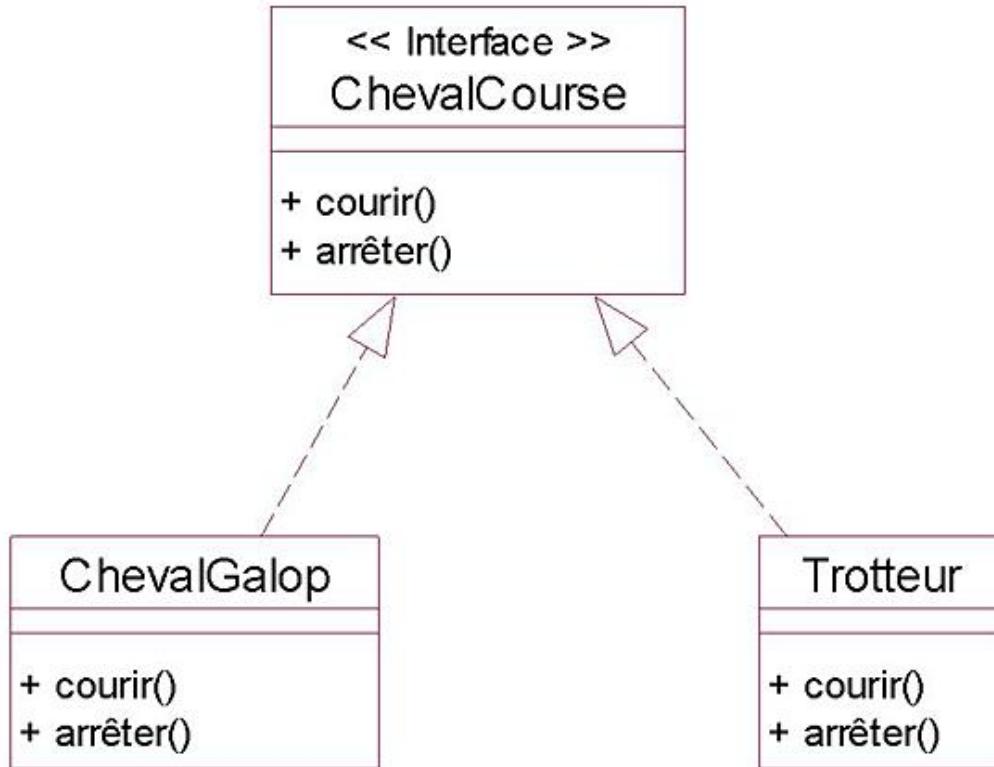


Figure 6.46 - Interface et sous-classes distinctes de réalisation

La figure 6.45 peut également être représentée par une *lollipop* (une *lollipop* est une sucette) (voir figure 6.47).

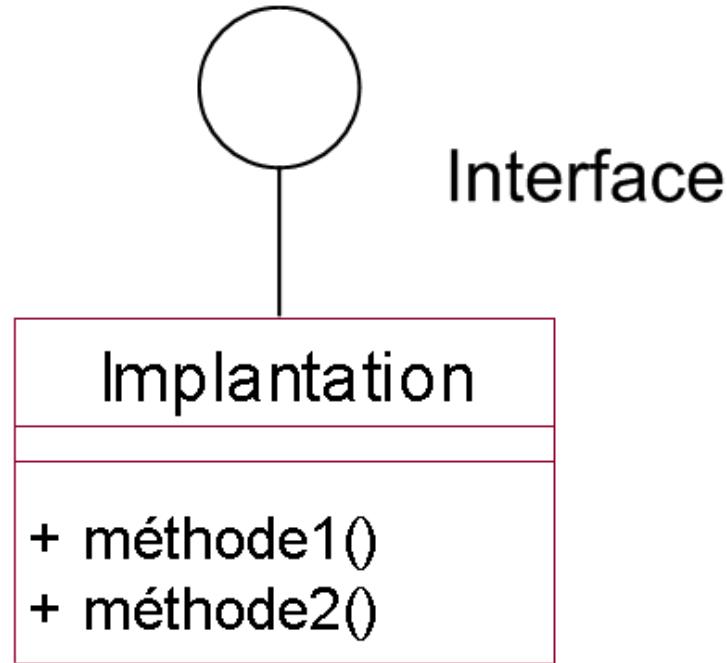


Figure 6.47 - Représentation lollipop d'une interface et de la relation de réalisation

► 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 6.48 illustre l'association de dépendance entre une classe et une interface, en reprenant la représentation de la figure 6.45 ou la représentation *lollipop* de l'interface.

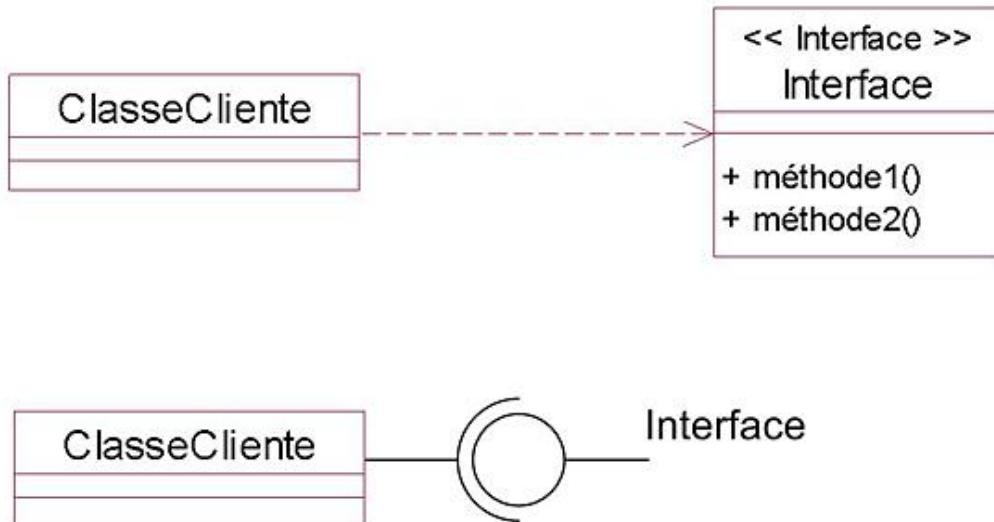


Figure 6.48 - Relation de dépendance entre une classe et une interface

- La relation de dépendance existe aussi entre deux classes. En effet, pour réaliser ces opérations, une classe peut dépendre d'une autre classe et non pas seulement d'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 6.49).

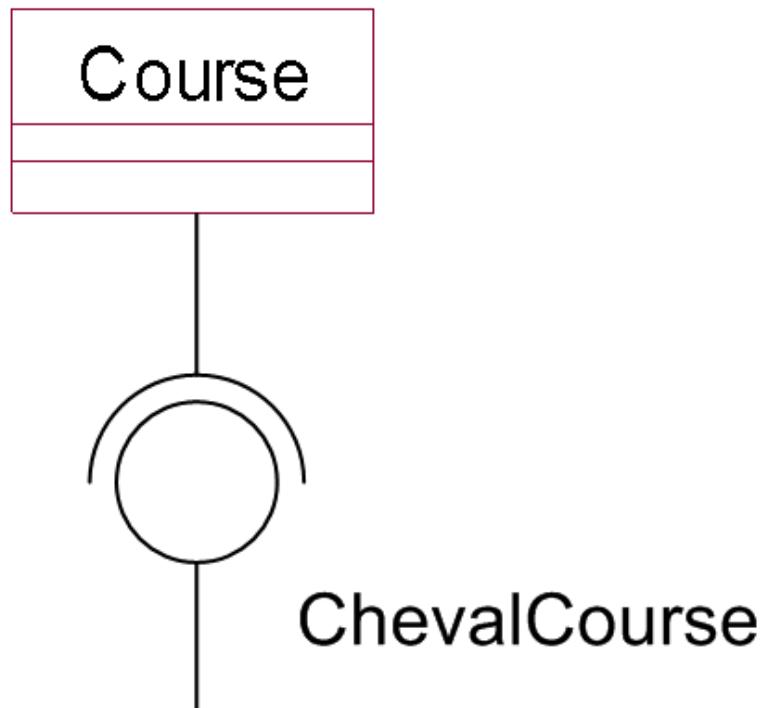


Figure 6.49 - Exemple de dépendance entre une classe et une interface

Le diagramme des objets ou instances

Le diagramme des classes est une représentation statique du système. Le diagramme des objets montre, 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 est de la forme :

```
nomInstance : nomClasse
```

Le nom de l'instance est optionnel.

La valeur d'un attribut est de la forme :

```
nomAttribut = valeurAttribut
```

Enfin, les liens entre instances sont représentés par de simples traits continus. Rappelons que ces liens sont les occurrences de relations interobjets.

Exemple

La figure 6.50 illustre un exemple de diagrammes d'objets. Le diagramme de classes dont il est issu étant représenté au-dessus.

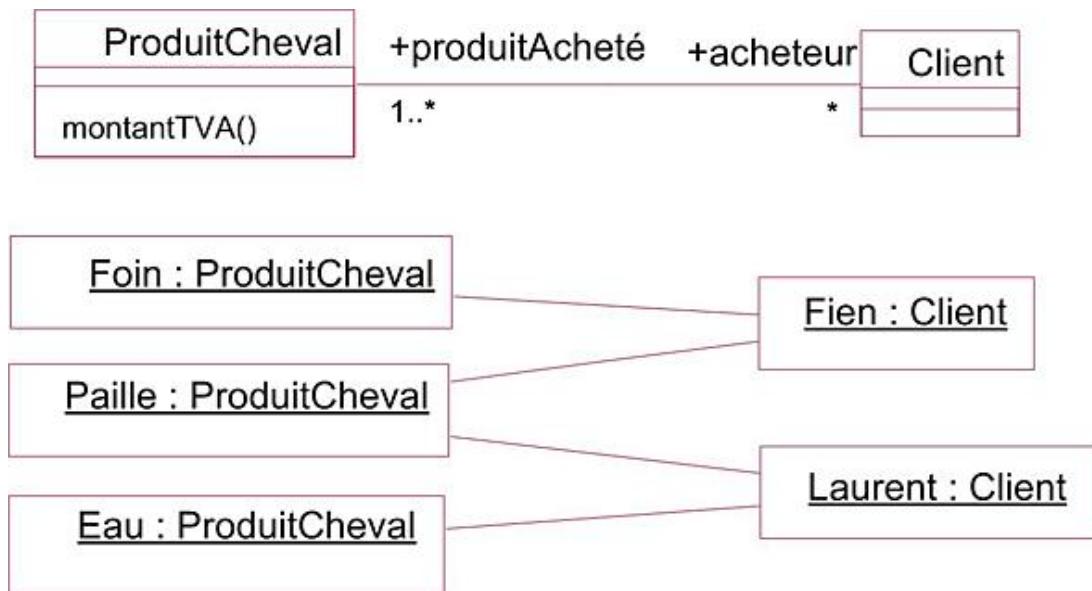


Figure 6.50 - Exemple de diagramme d'objets

Le diagramme de structure composite

1. Description d'un objet composé

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 des 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.

Nous considérons maintenant l'objet composé décrit par le diagramme des classes de la figure 6.51. Il possède un composant issu d'une composition forte et un autre issu d'une agrégation.

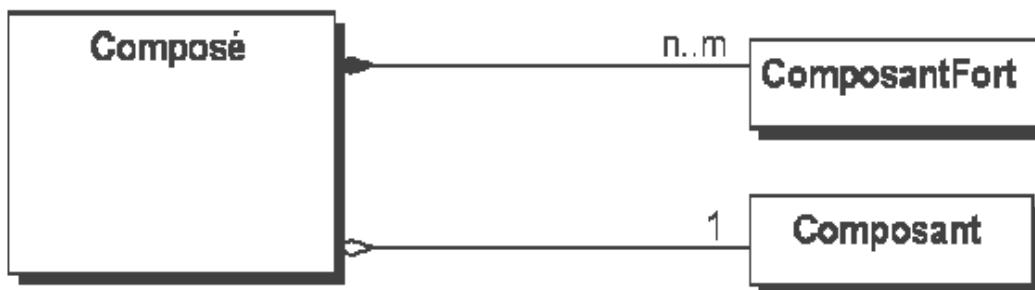


Figure 6.51 - Objet composé

La figure 6.52 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 un. Un composant issu d'une agrégation est représenté par une ligne en pointillé, un composant issu d'une composition forte est représenté par une ligne continue.

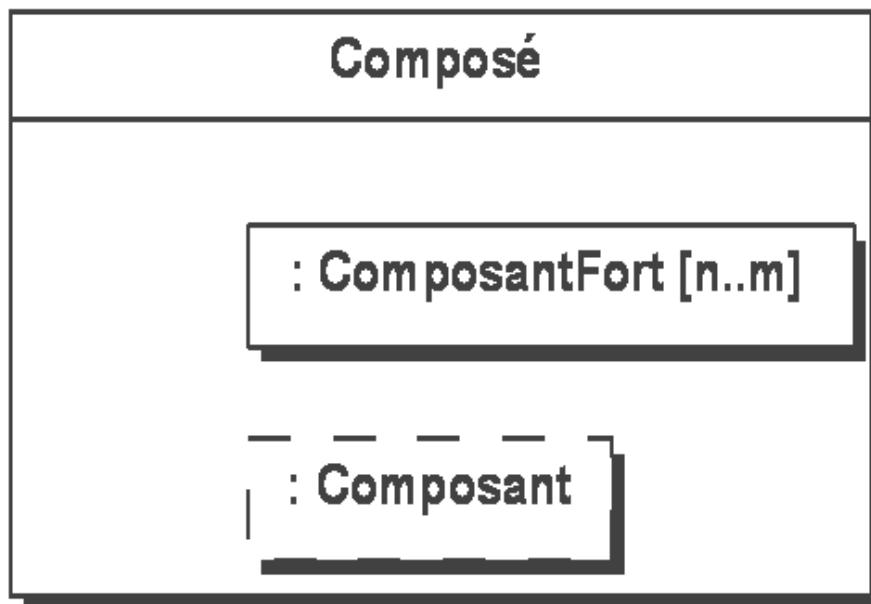


Figure 6.52 - Diagramme de structure composite

Exemple

La figure 6.53 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.

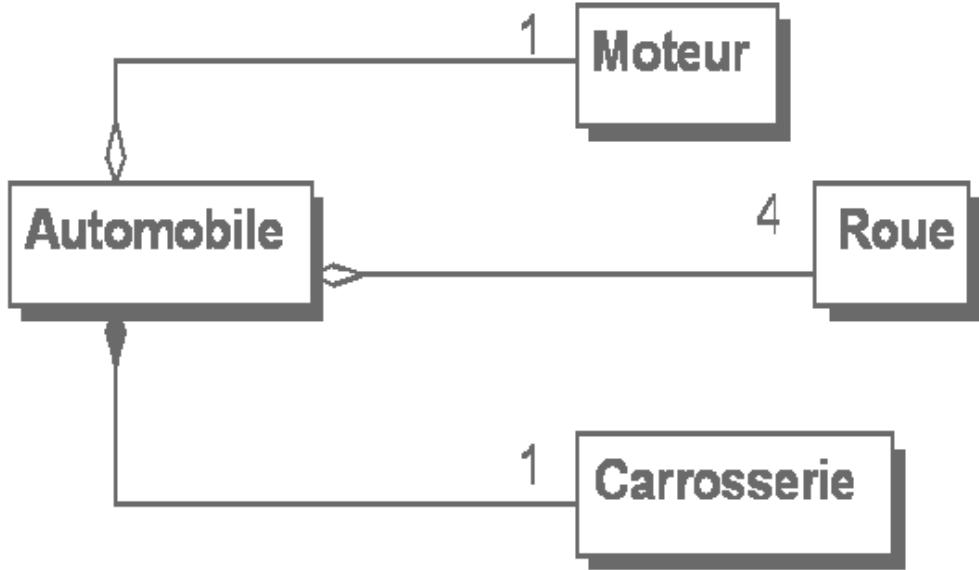
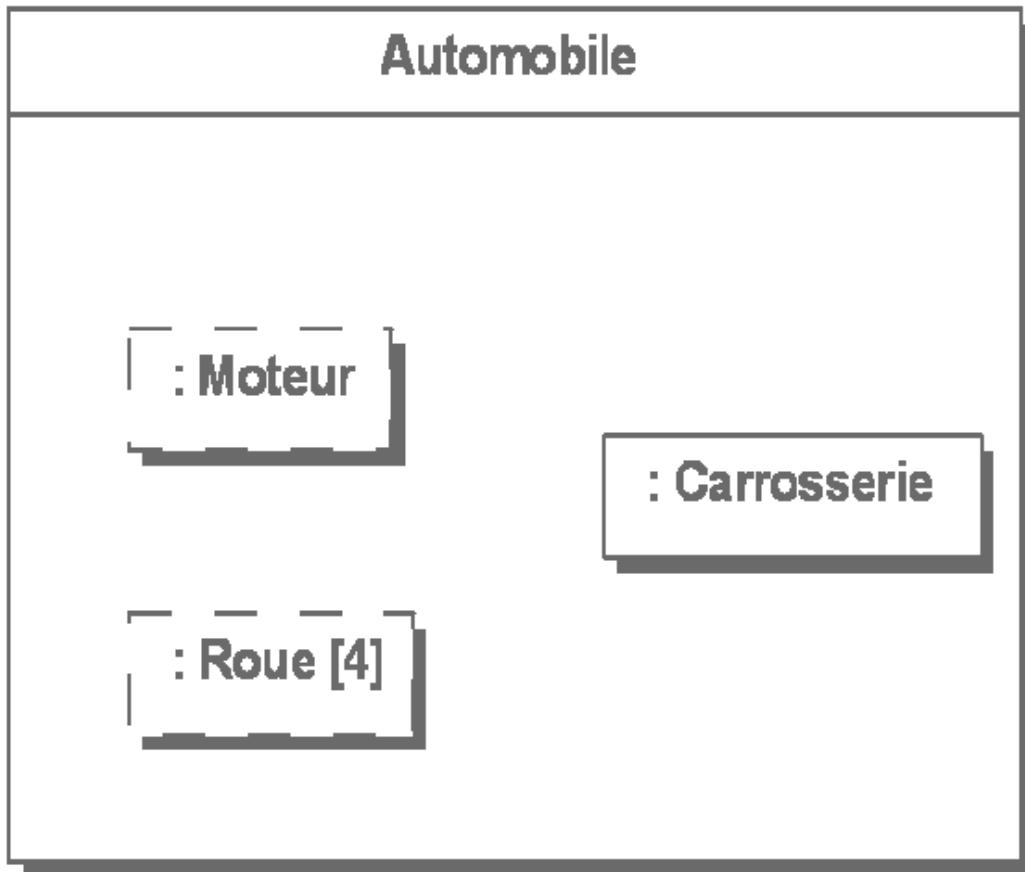


Figure 6.53 - Exemple de diagramme de structure composite

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 6.54 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.

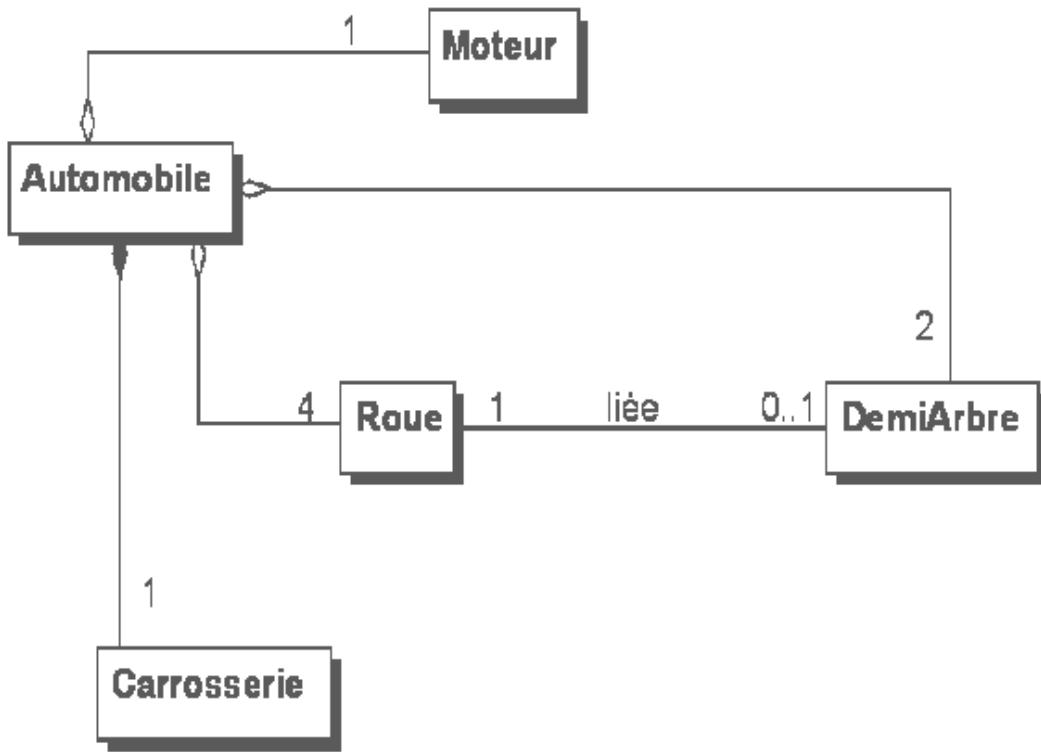


Figure 6.54 - Diagramme de classes de l'objet composé *Automobile* avec les demi arbres

La cardinalité de l'association *liée* est 0..1 à l'extrémité du demi arbre. En effet, la cardinalité vaut un pour les roues avant et vaut zéro 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 des 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 6.55 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.

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

Dans la figure 6.55, deux parties sont également introduites, correspondant à chaque demi arbre.

Entre chaque partie correspondant à une roue avant et à 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.

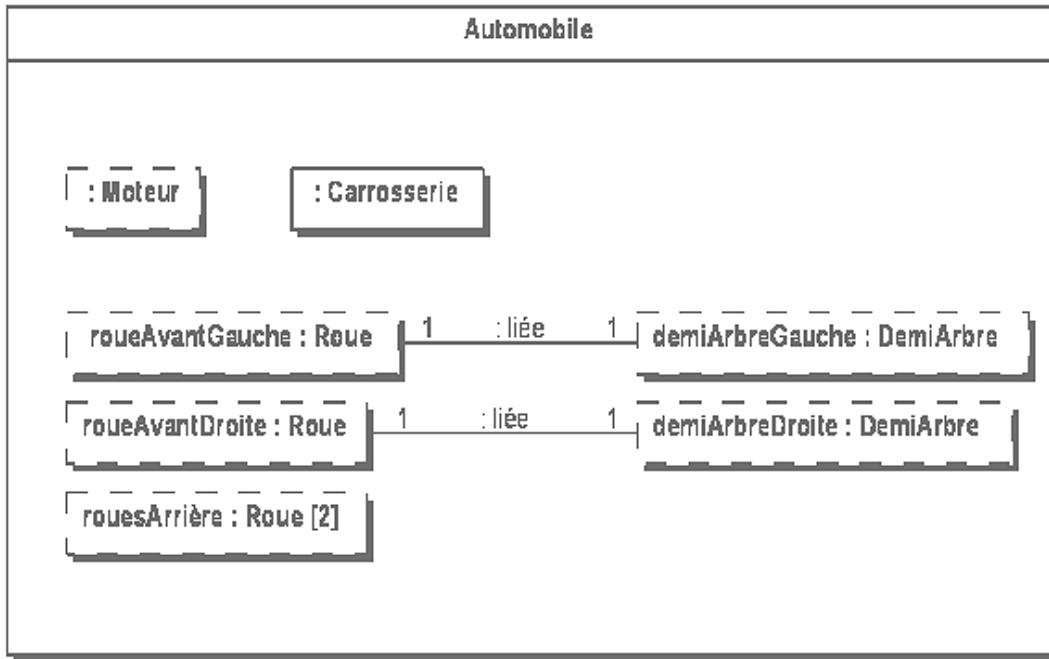


Figure 6.55 - Diagramme de structure composite avec rôles et connecteurs

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.

➤ 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 rôles différents, à l'instar de ce qui se fait pour les parties et les connecteurs.

La figure 6.56 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 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 ;
- que le moteur communique avec les demi arbres (transmission du mouvement) ;
- que 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*.

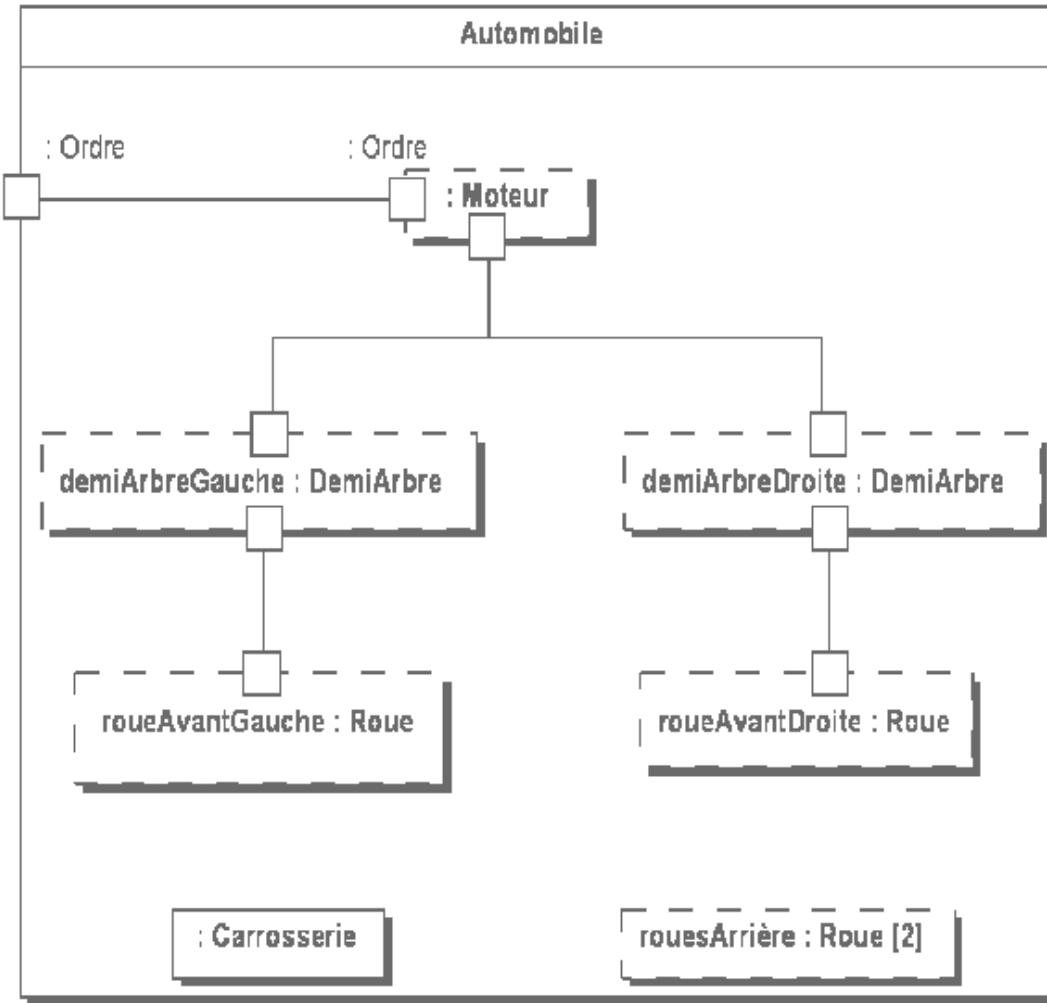


Figure 6.56 - Diagramme de structure composite introduisant des ports

2. Collaboration

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 6.57 montre le diagramme de classes de l'un de ces patterns, à savoir le pattern *Composite*.

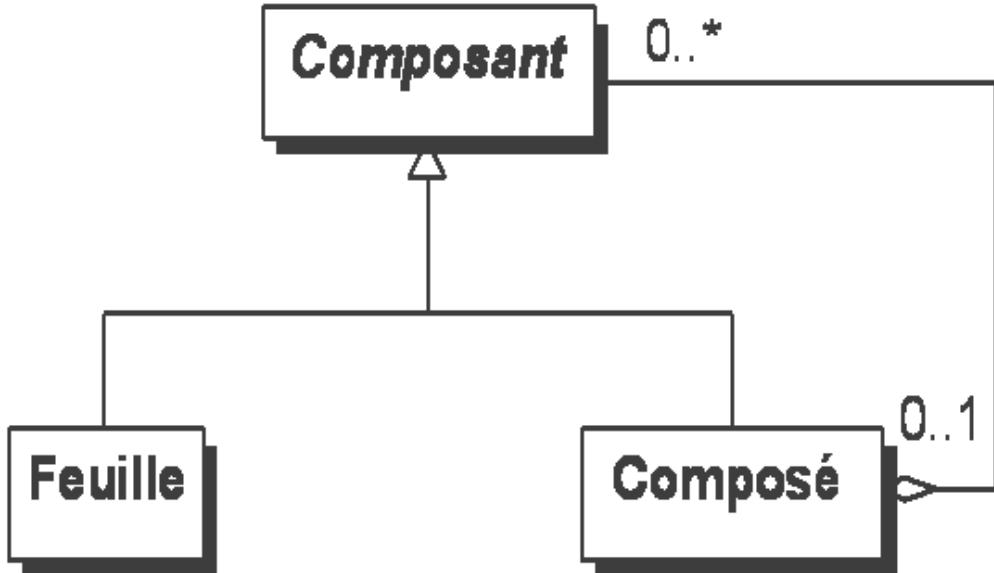


Figure 6.57 - Pattern Composite

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 6.58. 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).

 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 des classes mais à le compléter

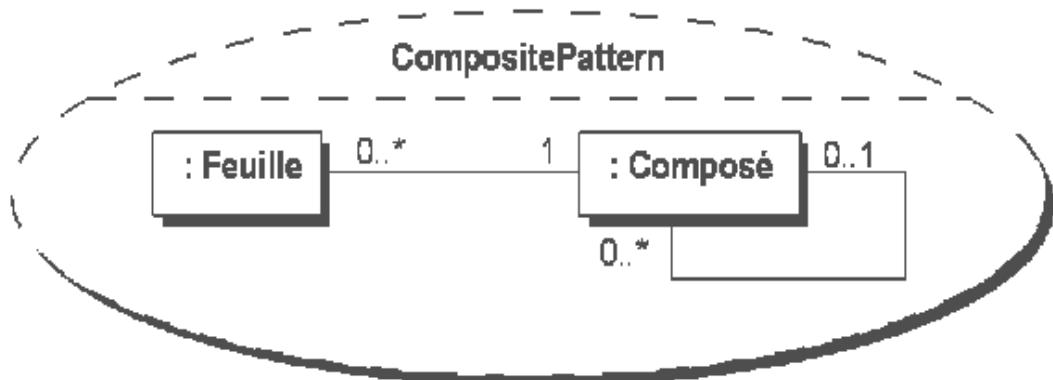


Figure 6.58 - Collaboration décrivant le pattern Composite

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 6.59 en prenant notre exemple de système de fichiers comme application de la collaboration du pattern Composite.

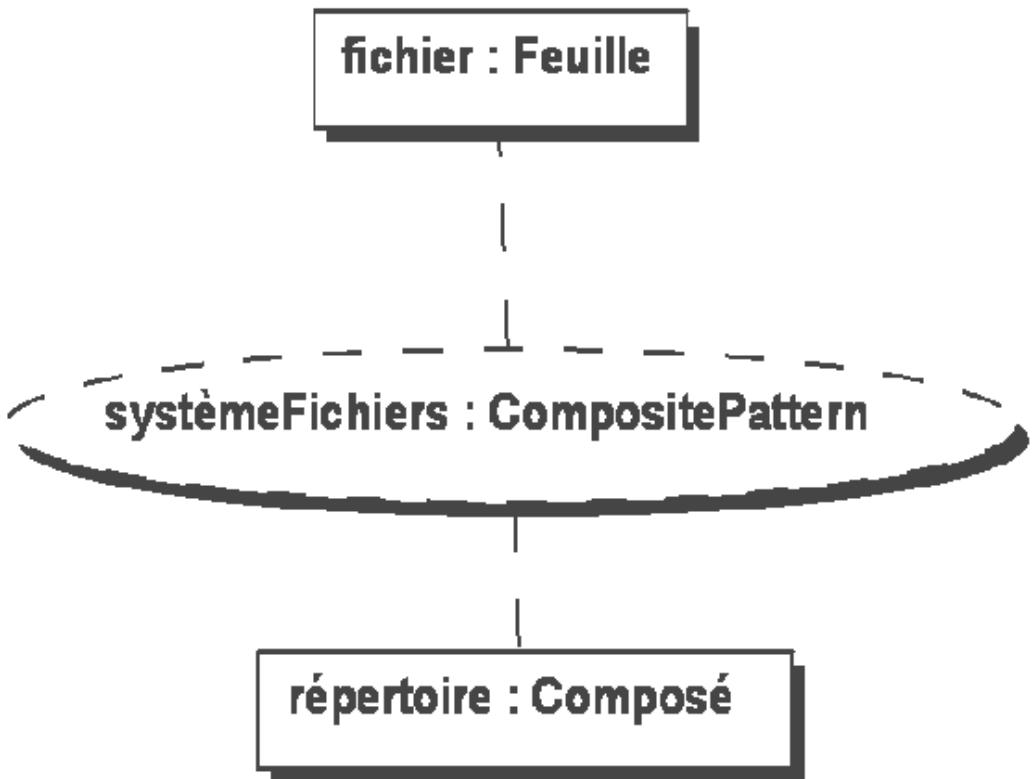


Figure 6.59 - Application de la collaboration Composite au système de fichiers.

Conclusion

Dans ce chapitre, nous avons étudié le diagramme des classes. Les classes décrivent les attributs et les méthodes de leurs instances, ainsi que les attributs et méthodes de classe.

Les associations entre objets sont obligatoires lors de la conception d'un diagramme de classes. Elles constituent le socle de l'interaction des instances par les envois de message lorsque le système est actif.

Les relations d'héritage entre les classes sont également indispensables. Elles favorisent la factorisation d'éléments communs et permettent ainsi de réduire conséquemment la taille du diagramme.

Enfin, l'expression des contraintes en langage naturel ou en OCL conduit à enrichir encore la sémantique exprimée dans le diagramme.

Exercices

1. La hiérarchie des chevaux

Soit les classes *Jument*, *Étalon*, *Poulain*, *Pouliche*, *Cheval*, *Cheval mâle* et *Cheval femelle* ainsi que les associations père et mère. Établir la hiérarchie des classes en y faisant figurer les deux associations.

Utiliser les contraintes {incomplete}, {complete}, {disjoint} et {overlapping}.

Introduire la classe *Troupeau*. Établir l'association de composition entre cette classe et les classes déjà introduites.

2. Les produits pour chevaux

Modéliser les aspects statiques du texte suivant sous la forme d'un diagramme de classes.

Une centrale des chevaux vend différents types de produits pour chevaux : *produits d'entretien*, *nourriture*, *équipement* (pour monter le cheval), *ferrures*.

Une commande contient un ensemble de produits avec pour chacun d'eux, la quantité. Un devis est éventuellement établi avant le passage de la commande. En cas de rupture de stock, la commande peut engendrer plusieurs livraisons si le client le désire. Chaque livraison donne lieu à une facture.

Introduction

UML 2 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.

-
- En UML 1, les paquetages faisaient partie du diagramme de classes et regroupaient exclusivement des ensembles de classes.
-

Paquetage et diagramme de paquetage

Un paquetage est représenté par un dossier (voir figure 7.1). Il constitue un ensemble d'éléments de modélisation UML.

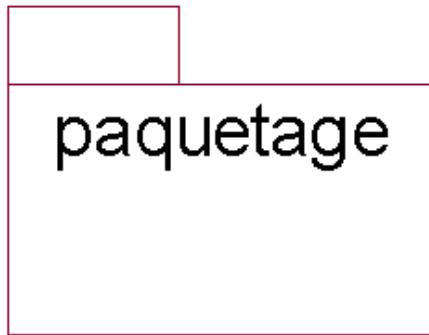


Figure 7.1 - Représentation graphique d'un paquetage

Exemple

Le paquetage regroupant les différents éléments de modélisation d'un élevage de chevaux est illustré à la figure 7.2.

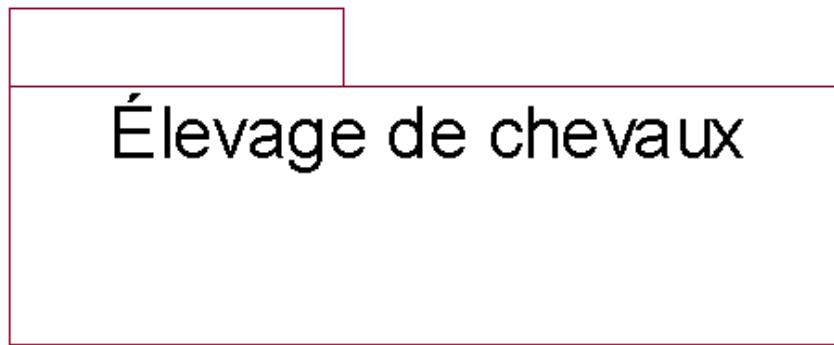


Figure 7.2 - Exemple de paquetage

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 7.3).

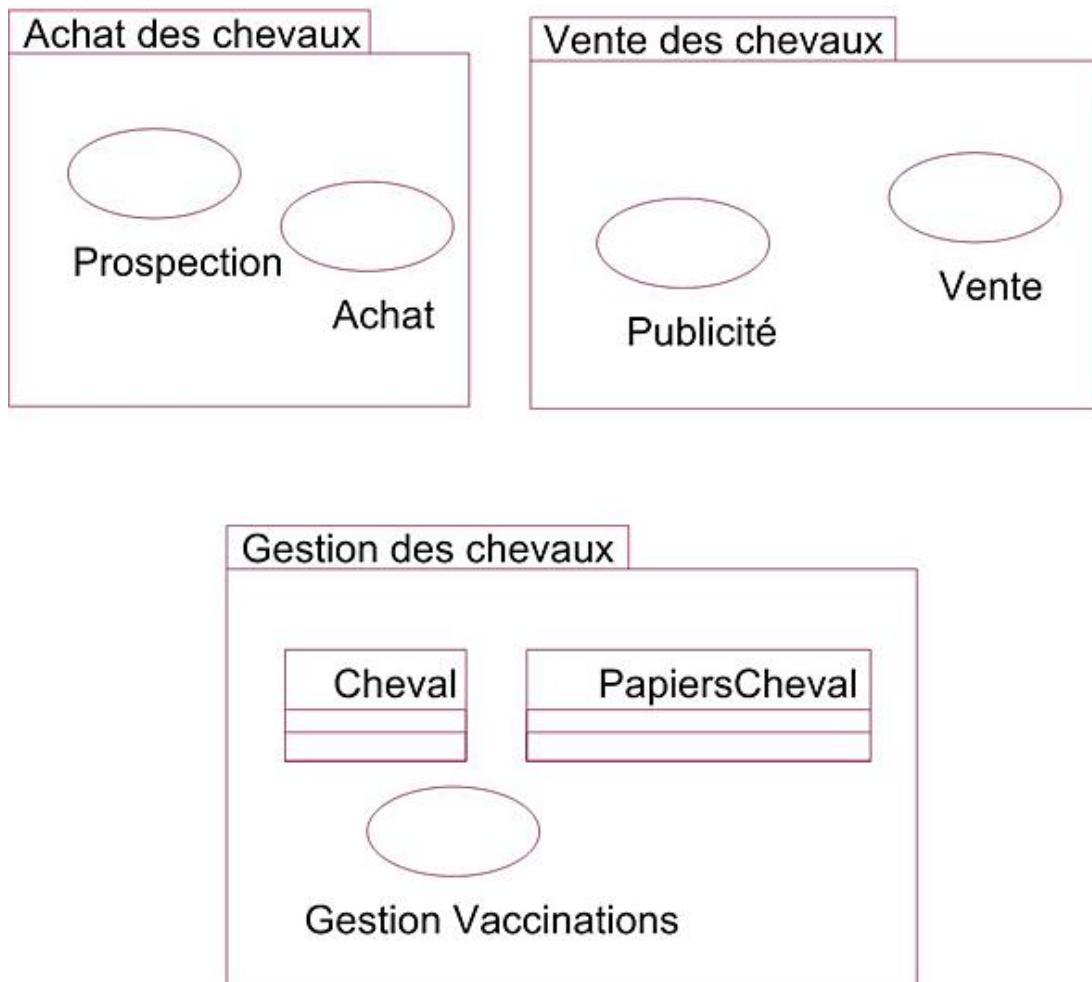


Figure 7.3 - Exemple du diagramme de paquetage représentant l'élevage de chevaux

➤ Chaque élément de la modélisation d'un système appartient à un seul paquetage. Nous verrons plus loin comment il peut être partagé par plusieurs paquetages.

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 7.4 illustre le paquetage "Élevage de chevaux" pour lequel l'encapsulation a été précisée.

Achat des chevaux



-Prospection



-Achat

Vente des chevaux



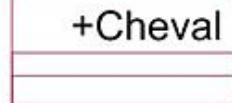
+Publicité



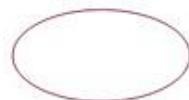
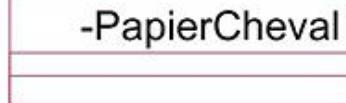
+Vente

Gestion des chevaux

+Cheval



-PapierCheval



-Gestion Vaccinations

Figure 7.4 - Exemple de diagramme de paquetage avec éléments encapsulés et visibles

Les associations entre les paquetages

Un élément de la modélisation ne peut être présent que dans un seul paquetage.

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

- L'association 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.
- L'association 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 spécifié visible dans le paquetage d'origine.

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

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

➤ En UML 1, seule la dépendance entre paquetages existait. En UML 2, celle-ci a été spécialisée et a donné lieu aux deux associations que nous venons de découvrir. Il reste cependant possible d'utiliser la dépendance simple d'UML 1.

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 associations d'importation sont illustrées à la figure 7.5.

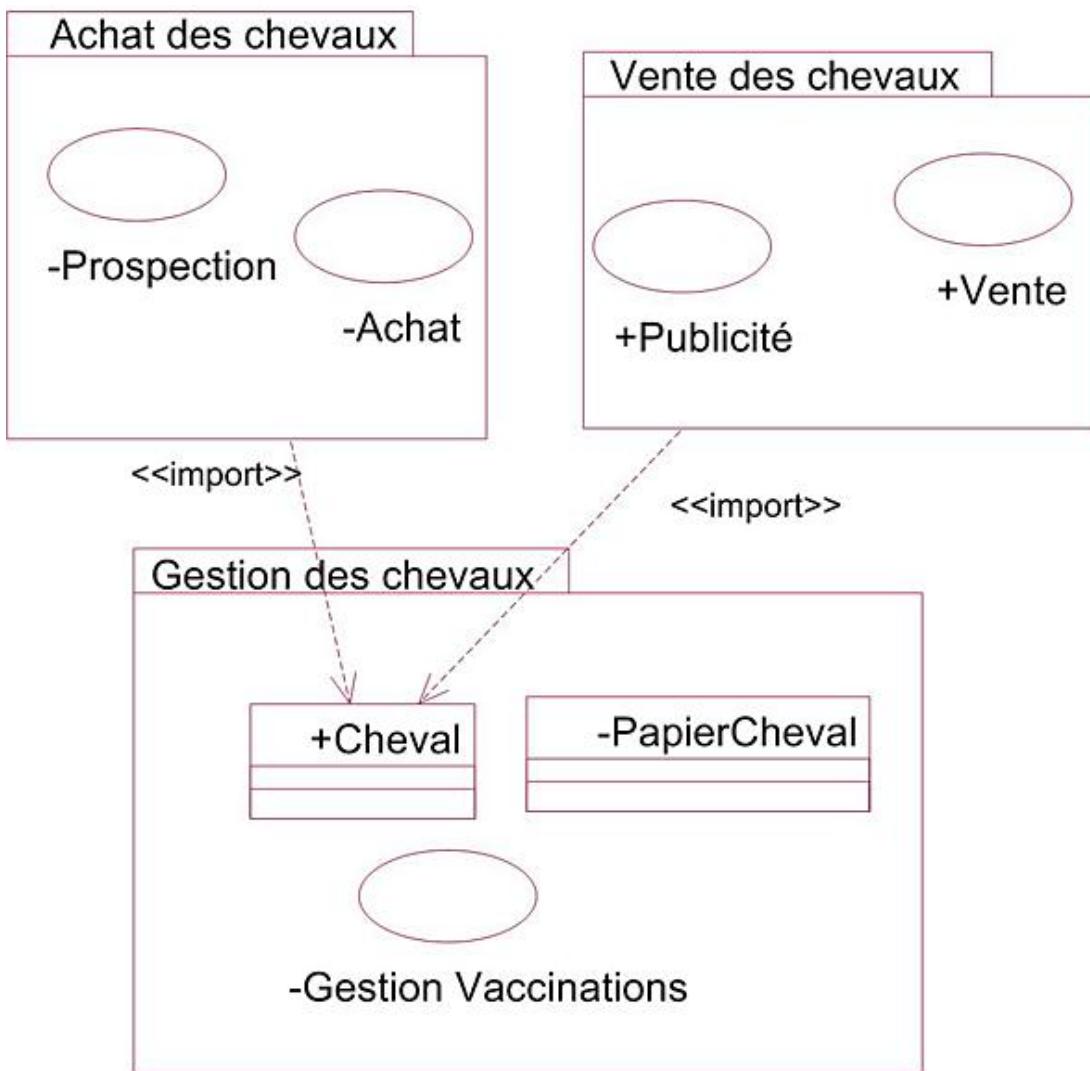


Figure 7.5 - Associations d'importation entre paquetages

Conclusion

Les paquetages sont utiles pour structurer la modélisation d'un système important. Un paquetage *système* contient les paquetages de plus haut niveau, eux-mêmes contenant d'autres paquetages et ainsi de suite, jusqu'aux éléments de base de la modélisation comme les classes ou les cas d'utilisation.

Cette structuration de la modélisation présente l'avantage de faire partie de la notation UML. Par conséquent, elle est standardisée.

Introduction

Après avoir étudié les interactions et les aspects statiques des objets du système, nous allons maintenant aborder leur cycle de vie. Le cycle de vie d'un objet représente les différentes étapes ou états que celui-ci va suivre pour concourir, au sein du système, à la réalisation d'un objectif. Un état correspond à un moment d'activité ou d'inactivité de l'objet. L'inactivité se produit lorsque l'objet attend que d'autres objets finissent une activité.

Nous étudierons la notion d'événement, signal qui fait changer l'objet d'état. Le changement d'état consiste à franchir une transition.

Le diagramme d'états-transitions illustre l'ensemble des états du cycle de vie d'un objet séparés par des transitions. Chaque transition est associée à un événement.

Nous étudierons en détail la notion de signal ainsi que la possibilité d'en envoyer à différents moments. Nous verrons qu'il est possible d'associer des conditions au franchissement des transitions et de gérer ainsi des alternatives.

La notion d'état composé sera examinée afin de simplifier l'écriture du diagramme d'états-transitions. Nous verrons qu'il est possible de disposer de sous-états évoluant en parallèle.

Enfin, nous présenterons le diagramme de *timing* qui décrit les changements d'état d'un objet lorsqu'ils sont exclusivement fonction du temps.

La notion d'état

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 suivant, nous en étudierons en détail la 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.

➤ Cette remarque est destinée plus particulièrement aux développeurs utilisant les langages à objets tels Java ou C++. Avec un tel langage, la plupart des programmes fonctionnent en monothread, c'est-à-dire en monotâche. Dans ce cas, un objet est inactif lorsqu'il ne dispose d'aucune méthode activée. Il devient actif lors de l'activation de l'une de ses méthodes. Cette activation correspond à la réception d'un signal en UML.

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. Ceux-ci correspondent à une phase de destruction de l'objet. Il arrive également qu'il n'y ait pas d'état final car un objet peut ne jamais être détruit.

Le changement d'état

1. La notion d'événement et de signal

Un événement est un fait qui déclenche le changement d'état. Il est lié à la réception d'un signal par l'objet. La réception d'un signal équivaut à la réception d'un message. Les envois et réceptions de messages ont été abordés dans le chapitre La modélisation de la dynamique consacré aux interactions.

- 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 hiérarchies.

- 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.

Exemple

La figure 8.1 montre la hiérarchie des signaux que peuvent recevoir le cavalier et son cheval. Il s'agit soit de signaux émis par le juge, soit par l'un des obstacles.

- Nous considérons qu'un obstacle peut émettre des signaux au même titre que le juge. Il ne faut pas oublier qu'en modélisation par objets, tout objet doit interagir avec les autres objets, fût-il passif dans le monde réel.

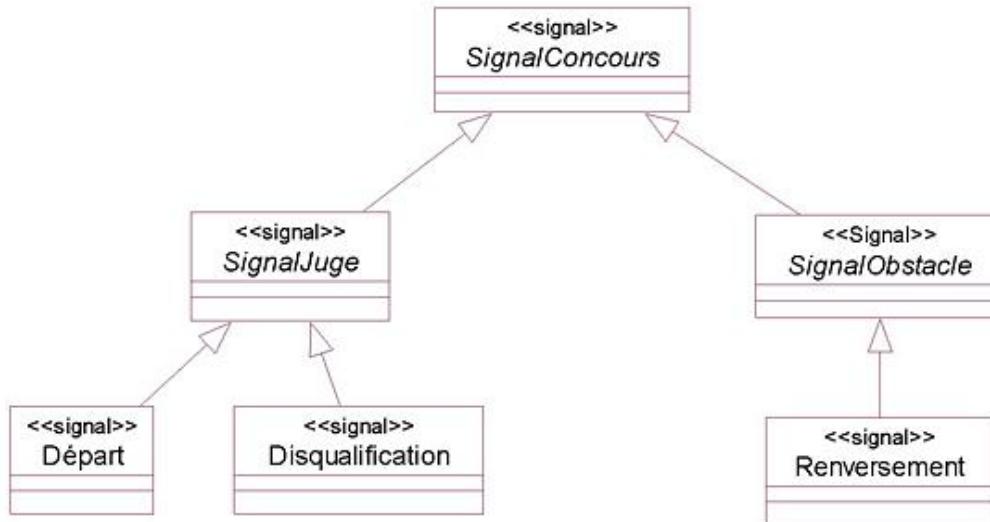


Figure 8.1 - Exemple de hiérarchie de classes représentant des signaux

- L'opération qui consiste à décrire une structure, ici un message par une classe, s'appelle la réification. Réifier, c'est transformer en chose. Dans l'approche par objets, c'est donc transformer en objet.

2. La 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 franchie.

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 s'il reçoit l'événement. 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.

Exemple

Lorsque le cavalier et le cheval reçoivent l'ordre de départ du juge, ils passent de l'état d'attente à l'état de course.

Une transition automatique n'est pas associée à un événement. Elle est franchie dès que l'objet a terminé l'activité liée à l'état d'origine. Dans ce cas, il est nécessaire d'associer une activité à l'état d'origine.

Exemple

Lorsque le cavalier et le cheval ont terminé le parcours, ils passent automatiquement dans l'état final.

Une transition réflexive possède le même état d'origine et de destination. Si la transition est associée à un événement, la réception de celui-ci ne fait pas changer l'état. Si la transition est réflexive et automatique, elle est alors utile pour réaliser une activité en boucle.

-
- Si un événement associé à une transition réflexive ne fait pas changer l'état de l'objet, sa réception peut provoquer une réaction comme l'appel d'une méthode de l'objet ou l'envoi d'un signal à d'autres objets.
-

Exemple

Tant que le cheval refuse de sauter un obstacle, il reste dans l'état de la course précédant cet obstacle. À chaque fois, le nombre de tentatives est augmenté de un.

L'élaboration du diagramme d'états-transitions

Le diagramme d'états-transitions représente le cycle de vie des instances d'une classe.

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.

1. La représentation graphique des éléments de base

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



Figure 8.2 - 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 (voir figure 8.3).



Figure 8.3 - Représentation graphique de l'état initial

Un état final correspond à une étape où l'objet n'est plus nécessaire dans le système et où il est détruit. Tous les objets n'ont pas d'état final. C'est notamment le cas des objets permanents dans le système.

Un état final est représenté par un point noir entouré d'un cercle (voir figure 8.4).



Figure 8.4 - Représentation graphique d'un état final

Une transition entre deux états est représentée par un trait droit fléché reliant ces deux états (voir figure 8.5). 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.

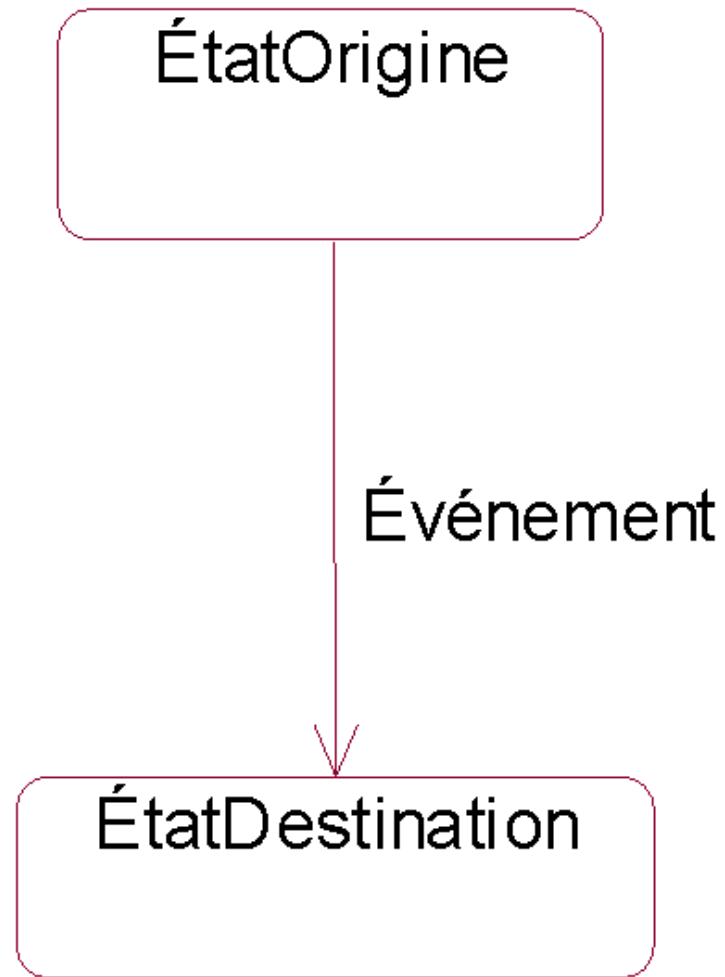


Figure 8.5 - Représentation graphique d'une transition

Une transition réflexive possède le même état d'origine et de destination (voir figure 8.6).

Événement

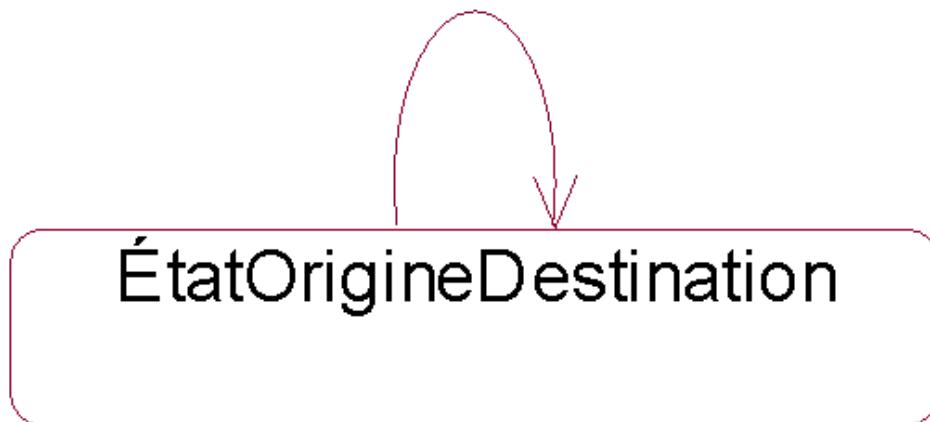


Figure 8.6 - Représentation graphique d'une transition réflexive

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 8.7 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.

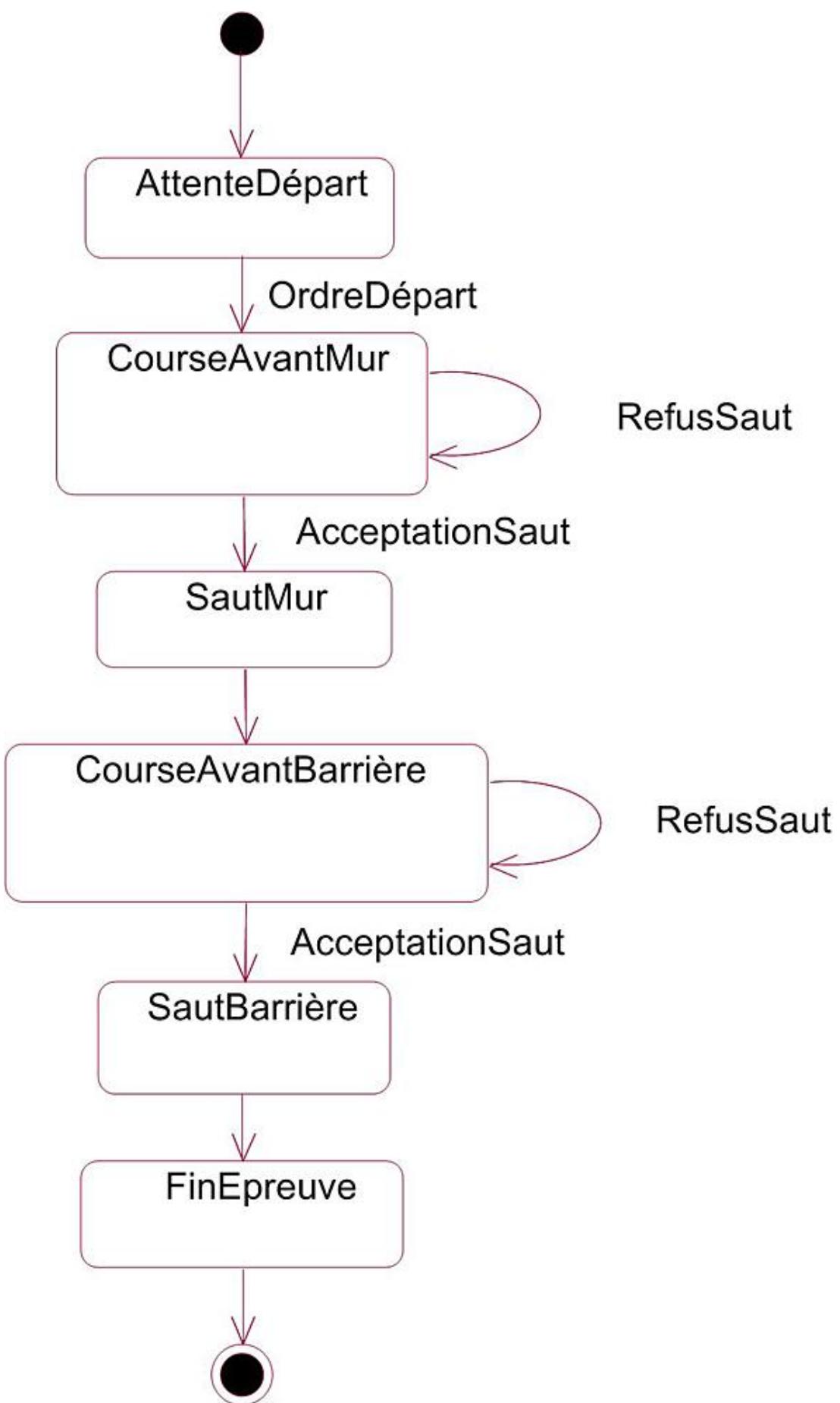


Figure 8.7 - Exemple de diagramme d'états-transitions

2. Les conditions de garde

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 réception 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 8.8).

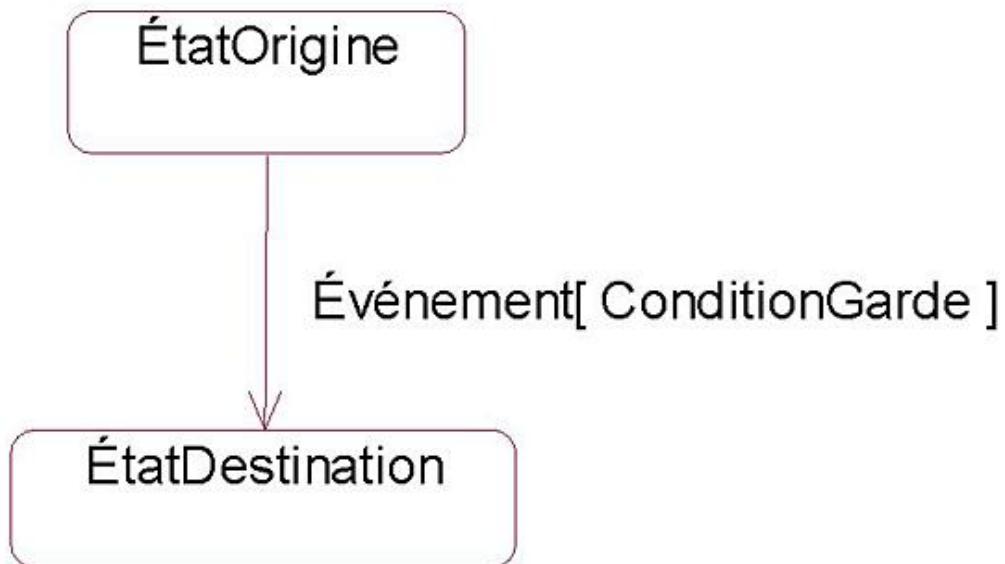


Figure 8.8 - Condition de garde

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 8.9 illustre la prise en compte de ce nombre maximal de refus en reprenant l'exemple de la figure 8.6 (seul le premier obstacle est montré).

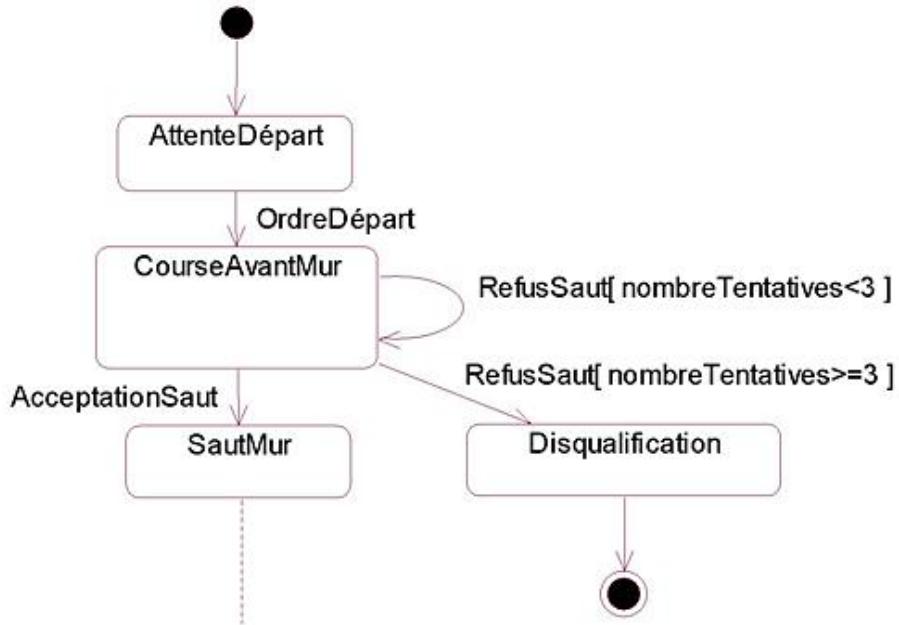


Figure 8.9 - Exemple d'utilisation des conditions de garde

3. Activités liées à un état ou à un franchissement de transition

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, créer ou détruire un objet, effectuer une opération, envoyer un signal à un autre objet ou à soi-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 8.10 illustre la représentation graphique de ces différentes possibilités. Une activité précédée du mot clé entry/ est exécutée lors de l'entrée dans l'état. Une activité précédée du nom d'un événement est exécutée si cet événement est reçu. Le mot clé do/ introduit l'activité réalisée pendant l'état. Une activité précédée du mot clé exit/ est exécutée lors de la sortie de l'état. L'envoi d'un signal est précédé d'un ^ suivi du nom du signal.

Il est également possible de spécifier une activité lors du franchissement d'une transition, qu'il faut faire précéder d'un /, et à la suite de l'événement et de la condition de garde, s'ils existent.

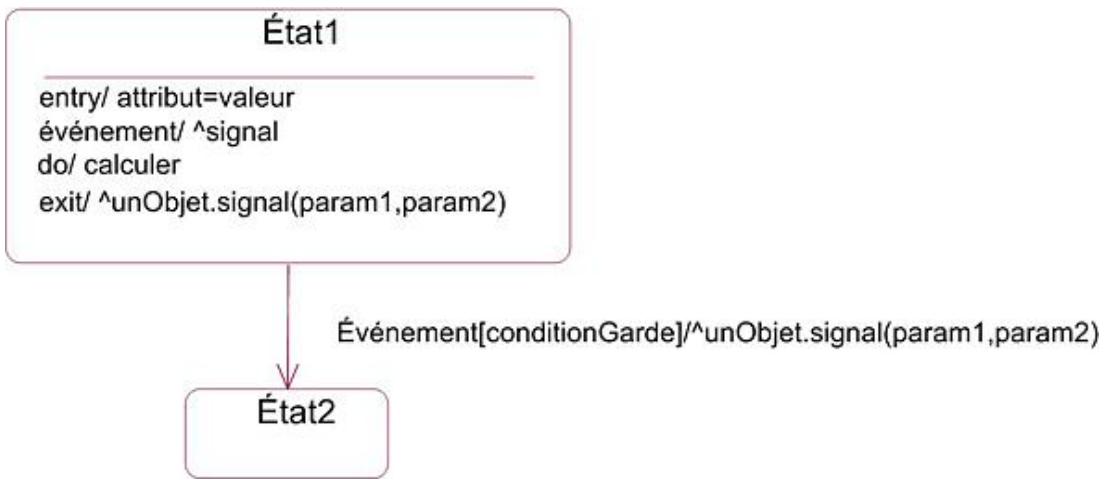


Figure 8.10 - Activités exécutées pendant un état ou lors du franchissement d'une transition

Exemple

La figure 8.11 illustre l'utilisation de ces 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 un, puis augmenté à chaque refus de sauter lors de la transition correspondante.

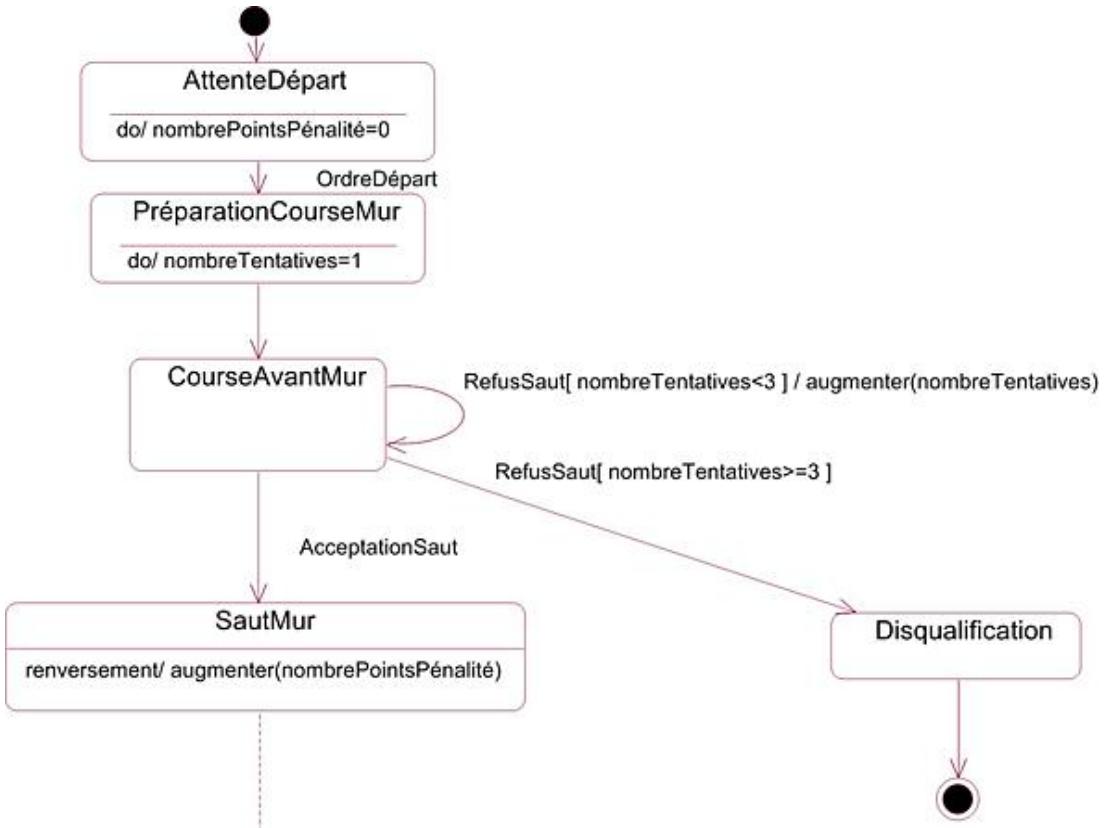


Figure 8.11 - Exemple d'activités au sein d'un état ou lors du franchissement d'une transition

4. États composés

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 8.12 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.

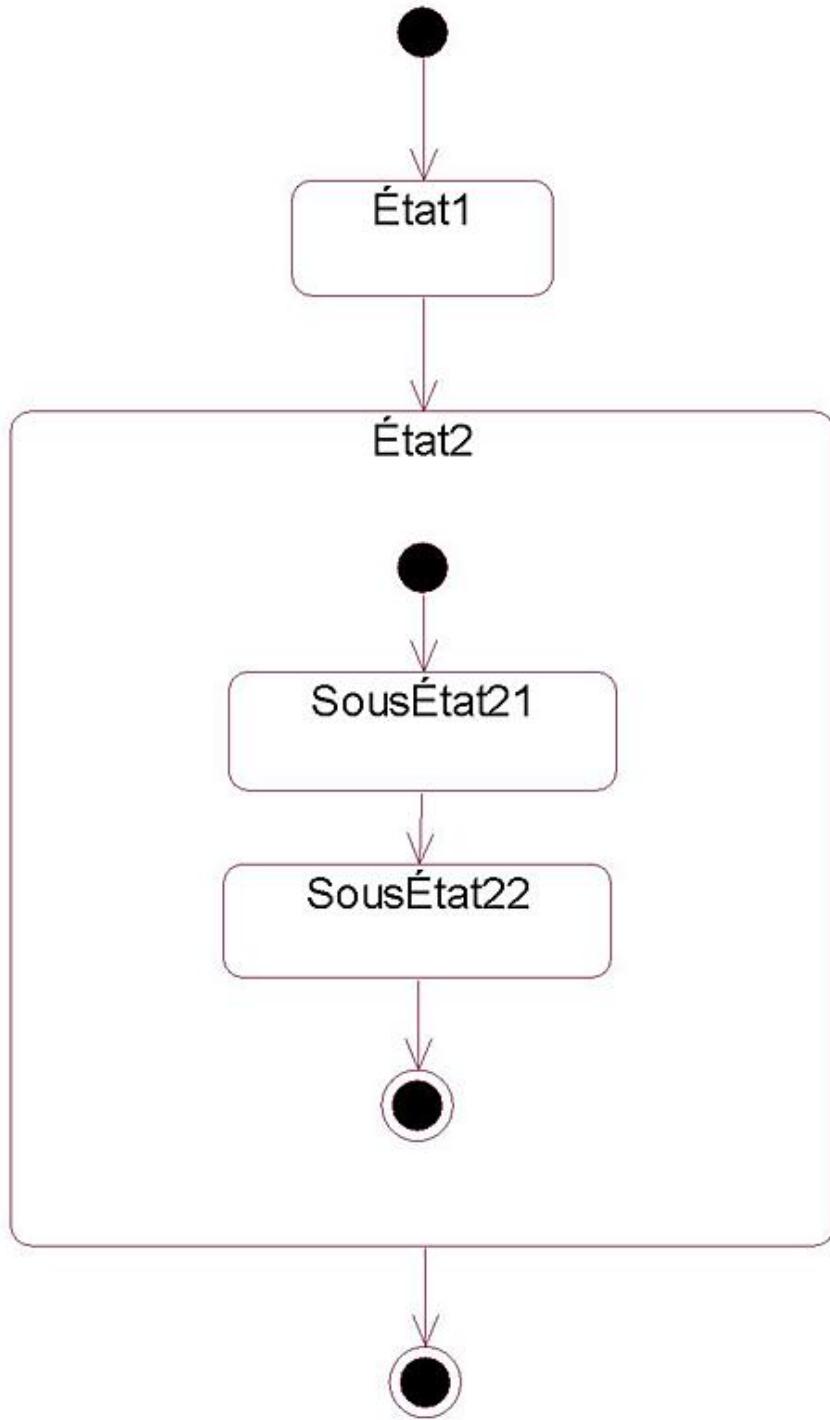


Figure 8.12 - État composé

Il est possible, lorsque 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 8.13 illustre ce sous-état spécial de mémoire.

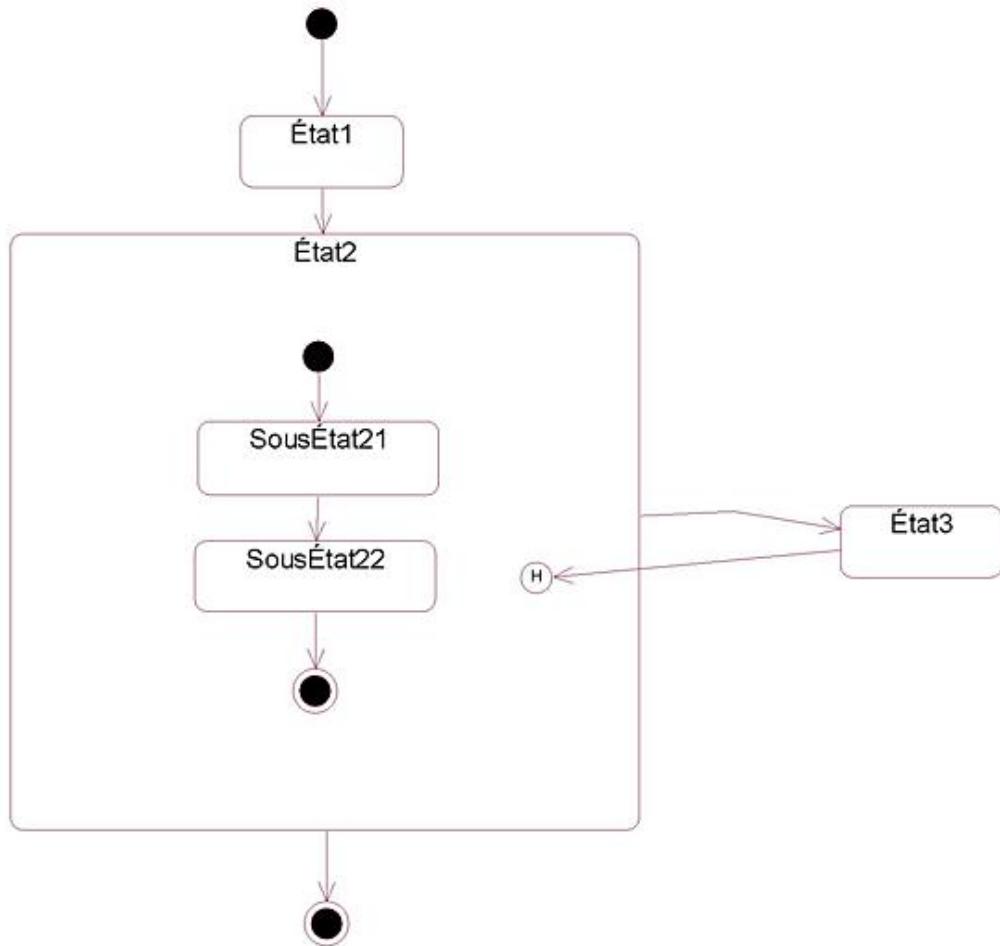


Figure 8.13 - Sous-état de mémoire

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

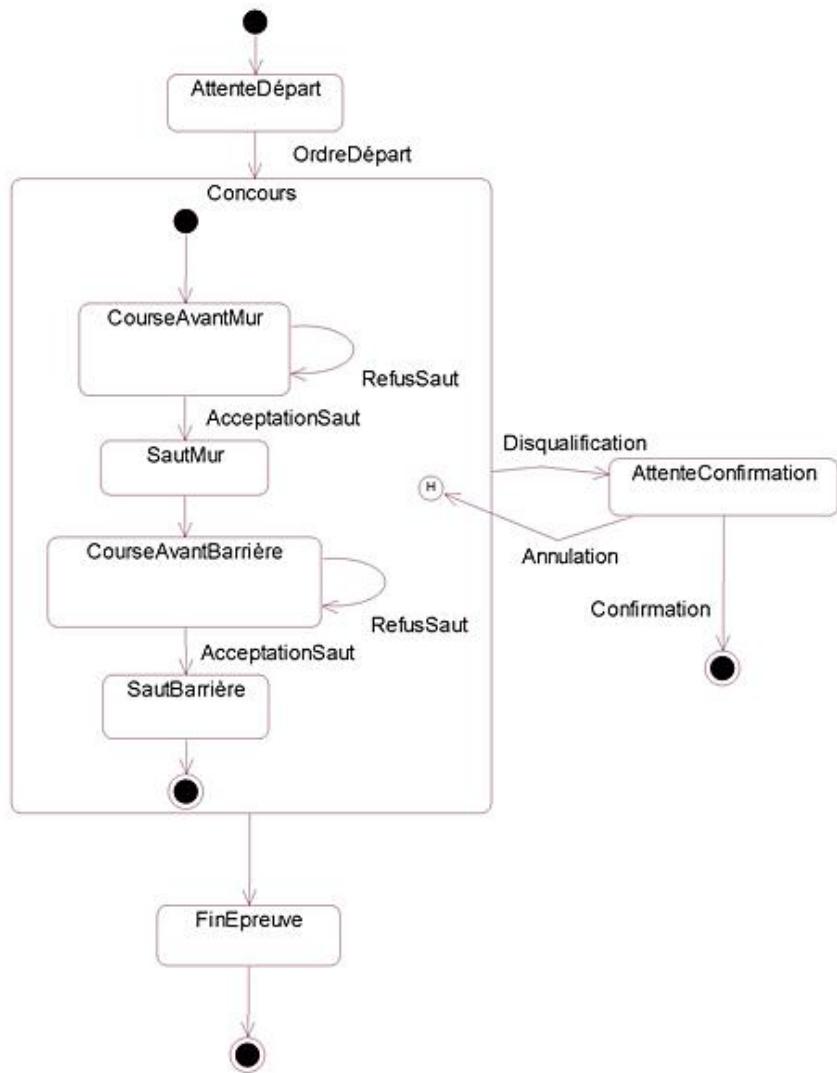


Figure 8.14 - Exemple de sous-états de mémoire

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 franchie, l'objet se trouve dans tous les sous-états de destination.

La transition de type *synchronisation* 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 8.15 fournit la représentation de ces deux types de transition.

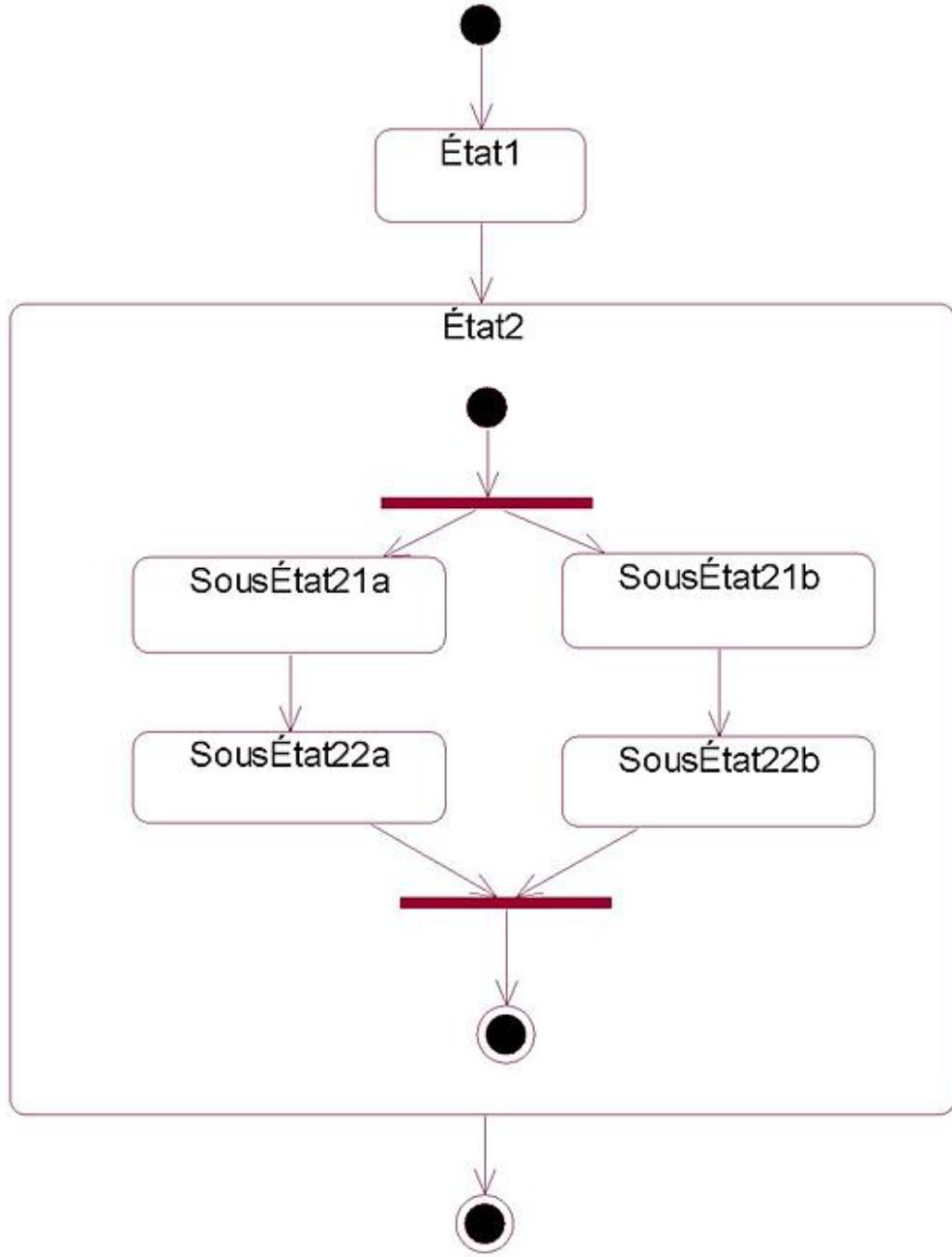


Figure 8.15 - Sous-états parallèles et transitions de fourche et de synchronisation

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 8.16. Dans l'état SautMur, une transition de fourche permet de distinguer les sous-états du cheval et du cavalier. 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. Une transition de synchronisation est franchie à la fin du saut.

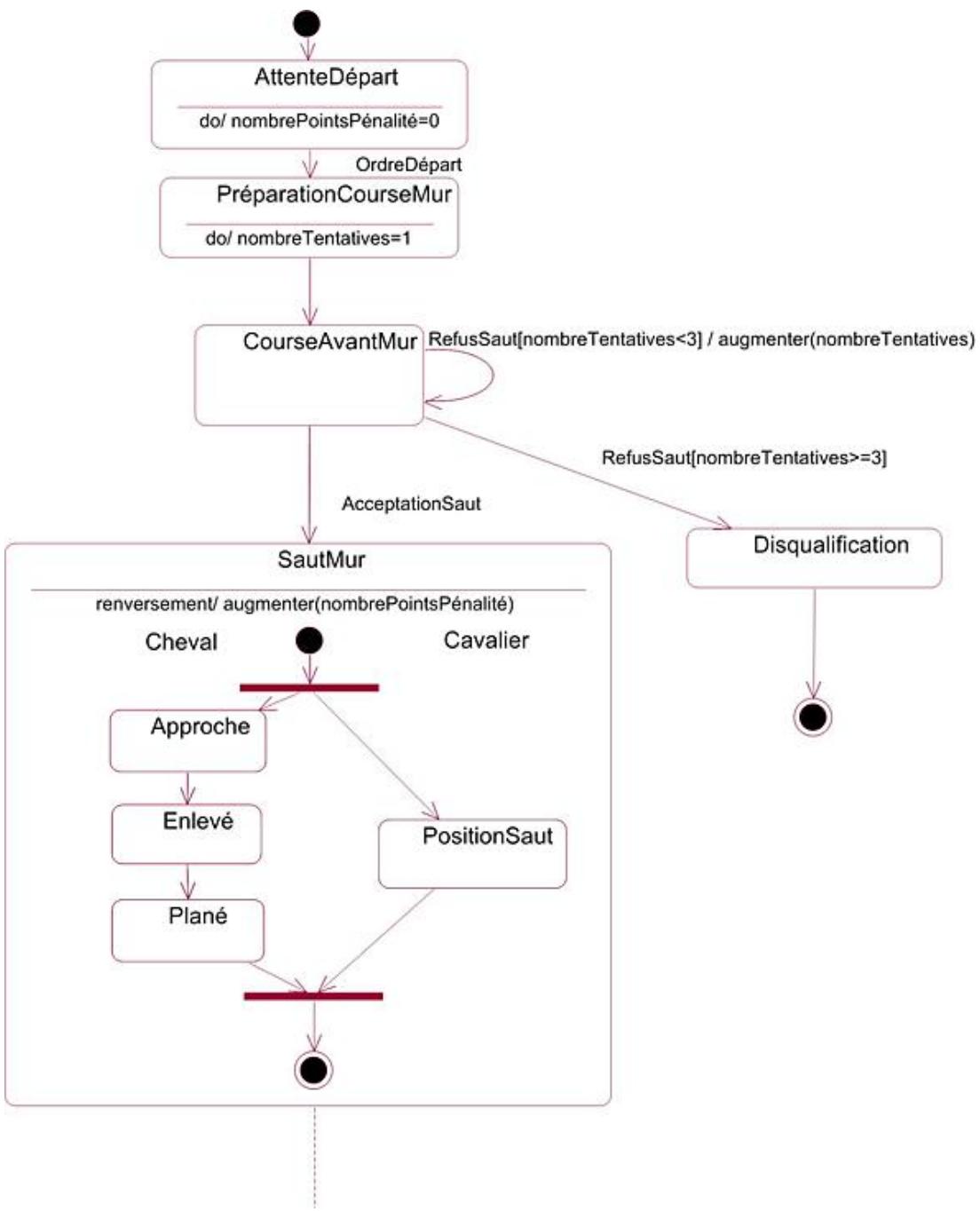


Figure 8.16 - Exemple de sous-états parallèles

Le diagramme de timing

Le diagramme de timing est introduit en UML 2 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 8.17 fournit la représentation graphique du diagramme de timing.

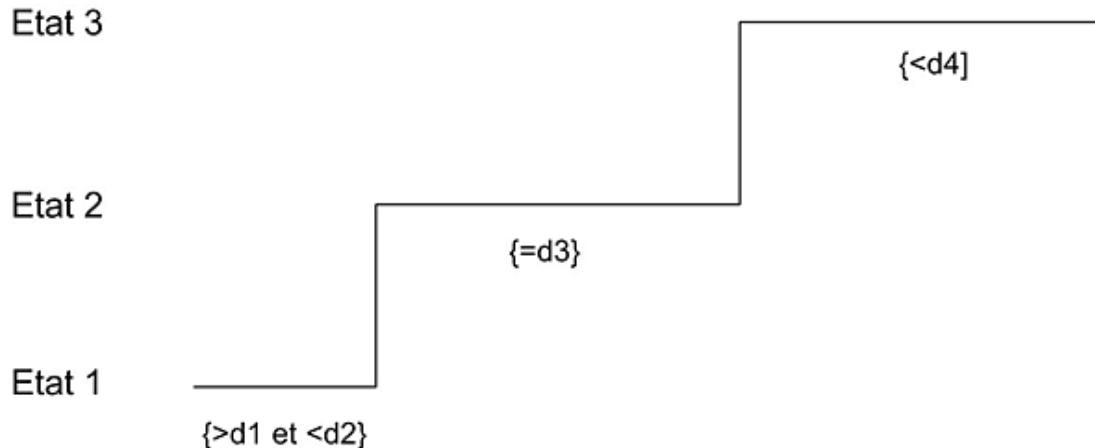


Figure 8.17 - Diagramme de timing

Exemple

Dans un concours d'obstacles, un cavalier doit s'imposer un temps maximum pour réaliser l'ensemble de l'épreuve, sinon il est éliminé. Il décompose lui-même ce temps sur chaque épreuve afin d'être sûr de réussir l'épreuve. La figure 8.18 illustre le diagramme de timing correspondant.

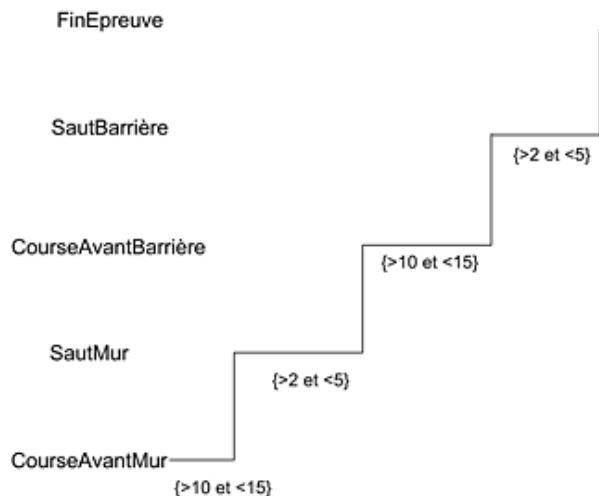


Figure 8.18 - Exemple de diagramme de timing

Conclusion

Le diagramme d'états-transitions décrit le cycle de vie des objets chargés d'assurer la dynamique du système. Cette description du cycle de vie est réalisée séparément pour chacun de ces objets.

Cette modélisation est très importante pour s'assurer que les objets puissent répondre aux interactions décrites dans les diagrammes de séquence et de communication que nous avons étudiés au chapitre La modélisation de la dynamique.

Exercices

1. Le ticket de course de tiercé

Dans quels états peut se trouver un ticket de course de tiercé ?

Construire le diagramme d'états-transitions d'une instance de la classe *Ticket*.

2. La course de chevaux

Dans quels états peut se trouver une course de chevaux ?

Construire le diagramme d'états-transitions d'une instance de la classe *Course*.

3. Le manège de bois

Décrire les différents états possibles d'un manège de chevaux de bois et construire le diagramme d'états-transitions correspondant.

Introduction

Le diagramme d'activités est basé sur le diagramme d'états-transitions étudié au chapitre précédent. Il s'agit d'une forme spécifique du diagramme d'états-transitions dans lequel chaque état est associé à une activité et toutes les transitions sont automatiques. Les transitions sont appelées enchaînements dans ce diagramme.

Le diagramme d'activités a ensuite été étendu pour décrire les activités de plusieurs objets. Les enchaînements entre les activités de différents objets peuvent ainsi être représentés, ce qui n'est pas possible avec le diagramme d'états-transitions. Nous verrons, pour chaque activité, comment désigner l'objet qui en est responsable grâce à la notion de travée.

Le diagramme d'activités offre des alternatives grâce aux conditions de garde. Il peut également contenir des enchaînements de type fourche et synchronisation pour gérer des activités parallèles.

Nous présenterons le diagramme de vue d'ensemble des interactions propre à UML 2.

Les activités et les enchaînements d’activité

1. Les activités

Une activité est une série d’actions. Une action consiste à affecter une valeur à un attribut, créer ou détruire un objet, effectuer une opération, envoyer un signal à un autre objet ou à soi-même, etc.

Sa représentation graphique est donnée à la figure 9.1.



Figure 9.1 - Représentation graphique d'une activité

Exemple

Nous reprenons l'exemple du chapitre *La modélisation des exigences sur l'achat d'une jument*. Le choix de la jument comme la vérification des vaccinations sont des exemples d'activité.

L'activité initiale est la première qui est exécutée. Elle est représentée par un point noir (voir figure 9.2).



Figure 9.2 - Représentation graphique de l'activité initiale

Une activité finale représente la fin de l'exécution de l'ensemble des activités d'un diagramme. Elle n'est pas forcément unique et n'est pas obligatoire.

Une activité finale est représentée par un point noir entouré d'un cercle (voir figure 9.3).



Figure 9.3 - Représentation graphique d'une activité finale

2. Les enchaînements d’activités

Un enchaînement d’activités est un lien orienté entre deux activités. Il est franchi dès que l’activité d’origine est terminée. Son franchissement conduit à l’enclenchement de l’activité de destination.

Il peut être simple, c'est-à-dire reliant deux activités. Sa représentation graphique est donnée à la figure 9.4.

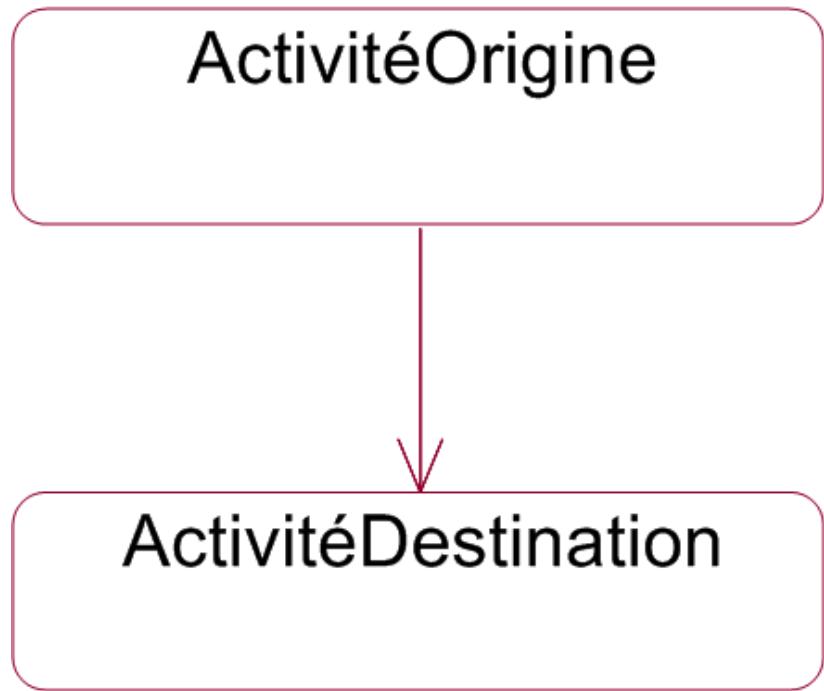


Figure 9.4 - Représentation graphique d'un enchaînement d'activités

Un enchaînement d'activités peut également être une alternative. Chaque branche de l'alternative est dotée d'une condition de garde exclusive des autres conditions. Rappelons que les conditions de garde ont été introduites au chapitre précédent.

La représentation graphique de l'alternative est illustrée à la figure 9.5.

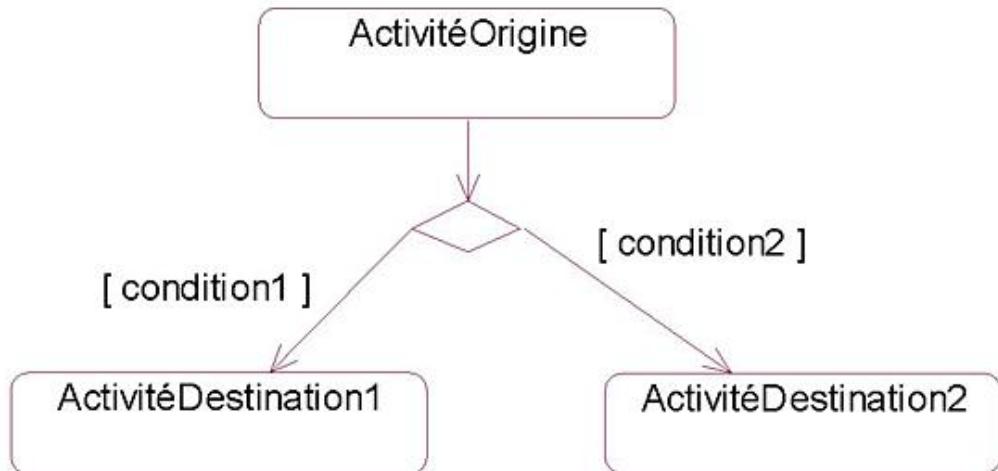


Figure 9.5 - Représentation graphique de l'alternative

Un enchaînement d'activités de type *fourche* possède également plusieurs activités de destination. Dès qu'il est franchi, toutes les activités de destination sont enclenchées en parallèle.

La représentation graphique de l'enchaînement de type *fourche* est donnée à la figure 9.6.

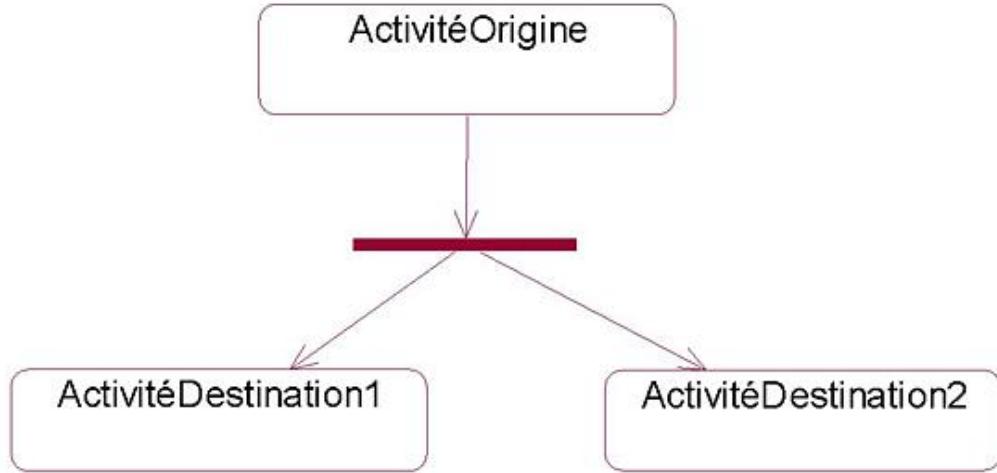


Figure 9.6 - Représentation graphique de l'enchaînement de type fourche

Un enchaînement d'activités de type *synchronisation* possède plusieurs activités d'origine et une seule activité de destination. Il faut que toutes les activités d'origine soient terminées pour qu'il soit franchi et que l'activité de destination soit enclenchée.

L'enchaînement de type *synchronisation* est illustré à la figure 9.7.

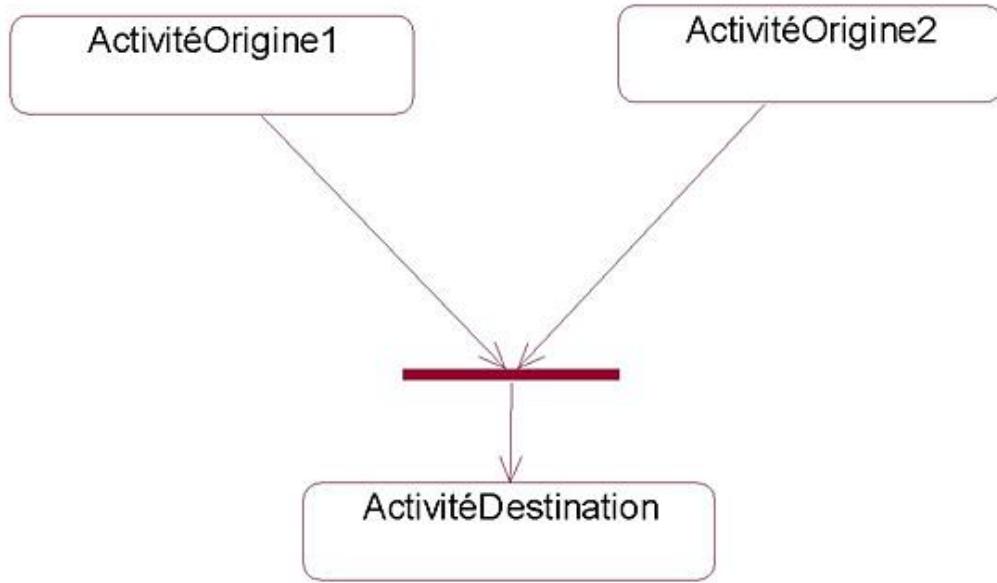


Figure 9.7 - Représentation graphique de l'enchaînement de type synchronisation

Exemple

Nous reprenons l'exemple de l'achat d'une jument. Le diagramme d'activités correspondant est décrit à la figure 9.8.

Le traitement des papiers et le transport de la jument sont traités en parallèle. En effet, l'acheteur peut très bien réaliser ces deux activités en même temps.

Les conditions de garde expriment les différentes alternatives. Il convient aussi de noter la présence de deux activités finales, l'une correspondant à l'abandon et l'autre à un achat réussi.

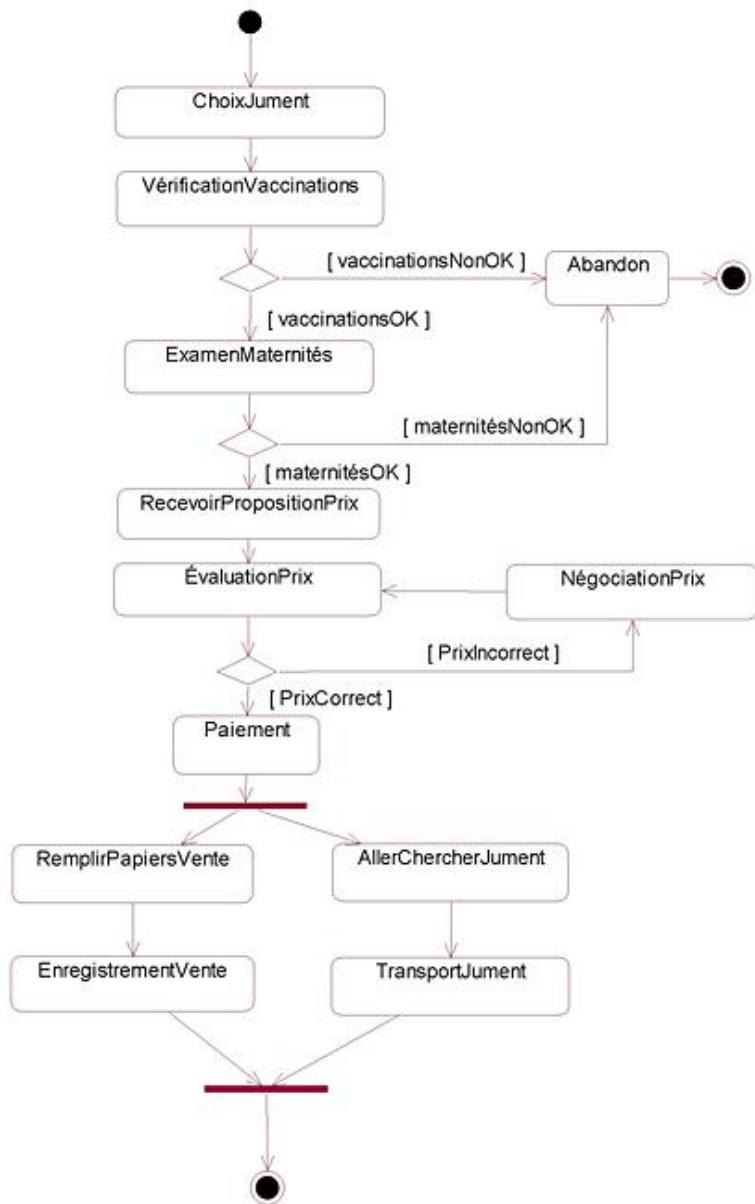


Figure 9.8 - Exemple de diagramme d'activités

Les couloirs

À la différence du diagramme d'états-transitions, le diagramme d'activités peut représenter les activités réalisées par plusieurs objets avec leurs enchaînements.

Pour cela, le diagramme est divisé en couloirs. À chaque couloir correspond l'objet responsable de la réalisation de toutes les activités contenues dans ce couloir.

La figure 9.9 illustre la représentation graphique des couloirs. Un enchaînement peut couper la ligne de séparation de deux couloirs pour montrer un changement d'objet entre l'activité d'origine et celle de destination.

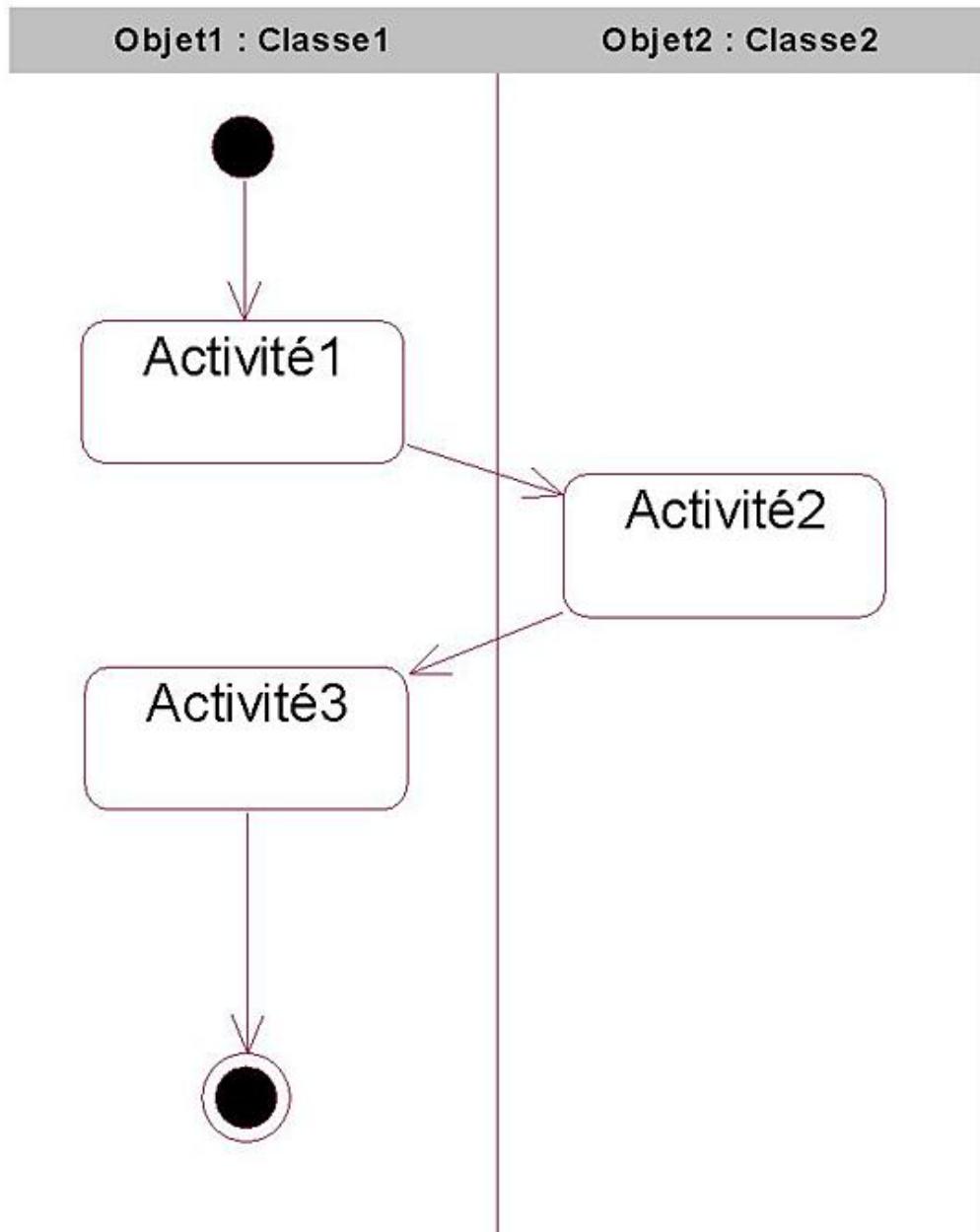


Figure 9.9 - Les couloirs d'un diagramme d'activités

Exemple

La figure 9.10 représente l'exemple de l'achat d'une jument où les activités relatives à l'acheteur et à l'élevage vendeur sont décrites.

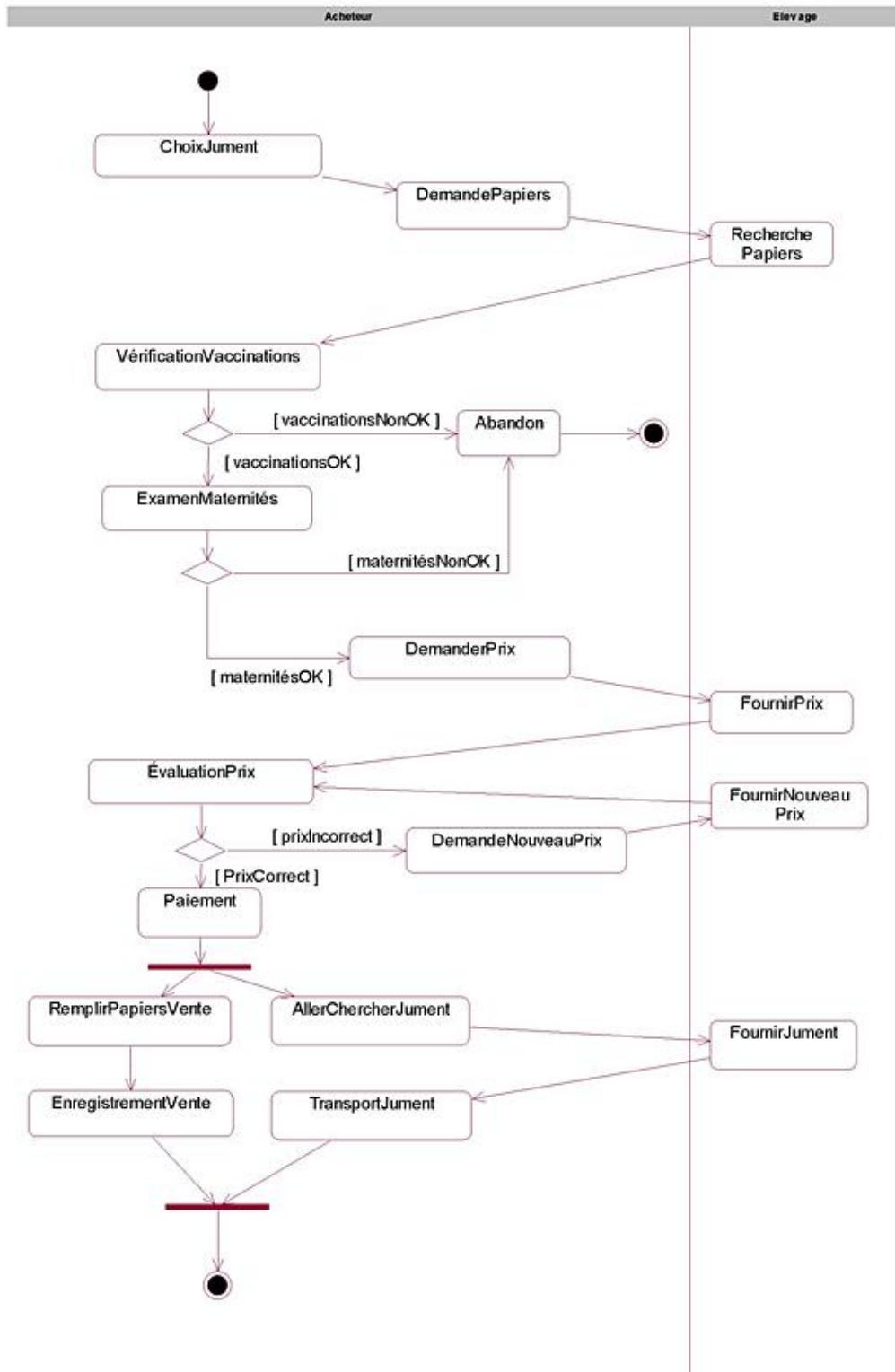


Figure 9.10 - Exemple de diagramme d'activités divisé en couloirs

Les activités composées

Une activité peut être composée d'autres activités. Dans ce cas, un diagramme d'activités spécifique en décrit la composition en sous-activités. Dans les diagrammes où elle est présente, une activité composée est représentée avec un symbole de fourche.

La figure 9.11 montre la composition d'une activité en sous-activités. La figure 9.12 illustre l'activité sans sa composition, telle qu'elle peut être utilisée au sein d'un diagramme d'activités.

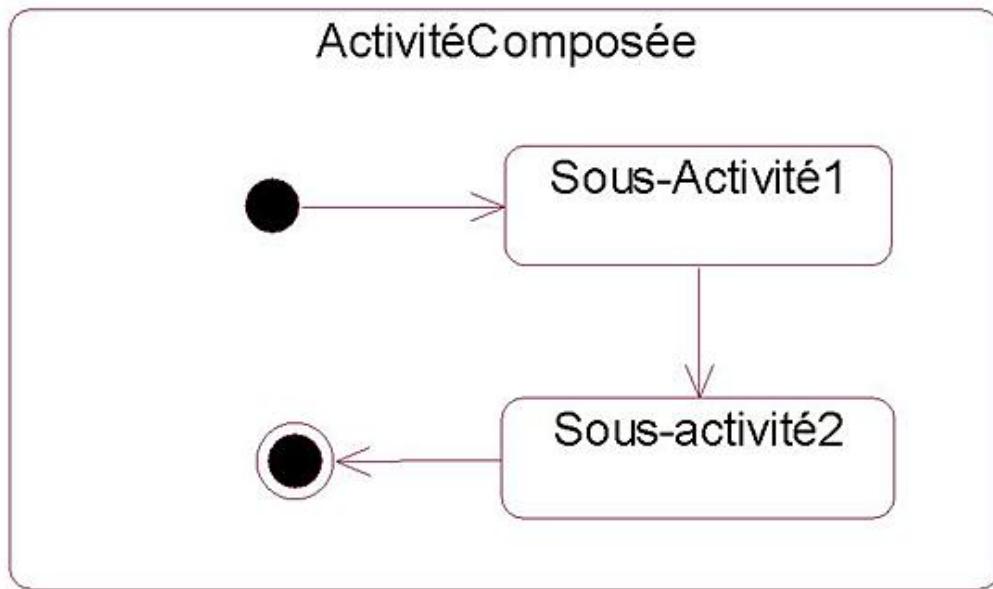


Figure 9.11 - Composition en sous-activités d'une activité

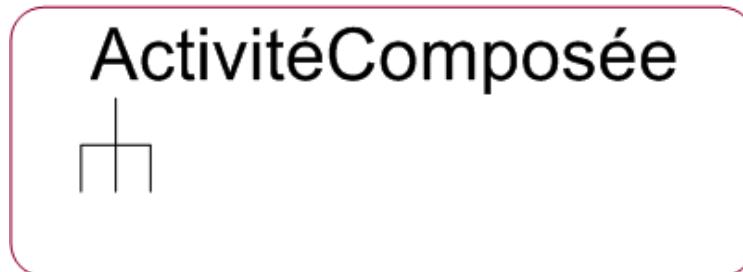


Figure 9.12 - Représentation d'une activité composée

Exemple

La figure 9.13 représente la gestion du paiement d'une jument en intégrant la négociation du prix. Cette gestion est incluse dans le diagramme global d'achat à la figure 9.14 en tant qu'activité composée où elle est, en conséquence, représentée avec une fourche.

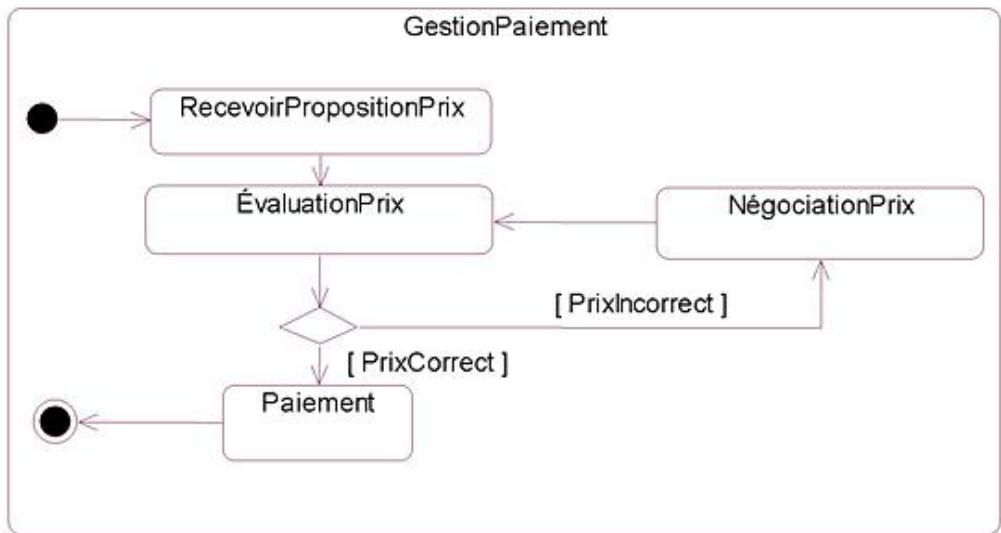


Figure 9.13 - Exemple d'activité composée

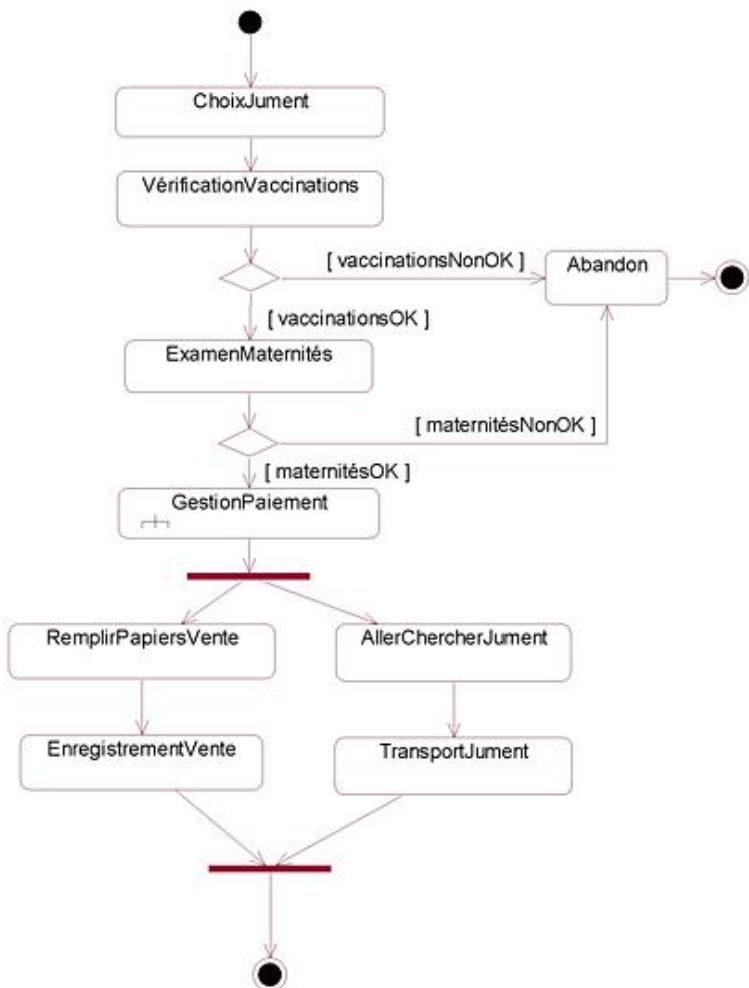


Figure 9.14 - Exemple d'inclusion d'une activité composée

Le diagramme de vue d'ensemble des interactions

Le diagramme de vue d'ensemble des interactions est spécifique à UML 2. Il s'agit d'un diagramme d'activités où chaque activité peut être décrite par un diagramme de séquence. La figure 9.15 montre la représentation graphique d'un tel diagramme.

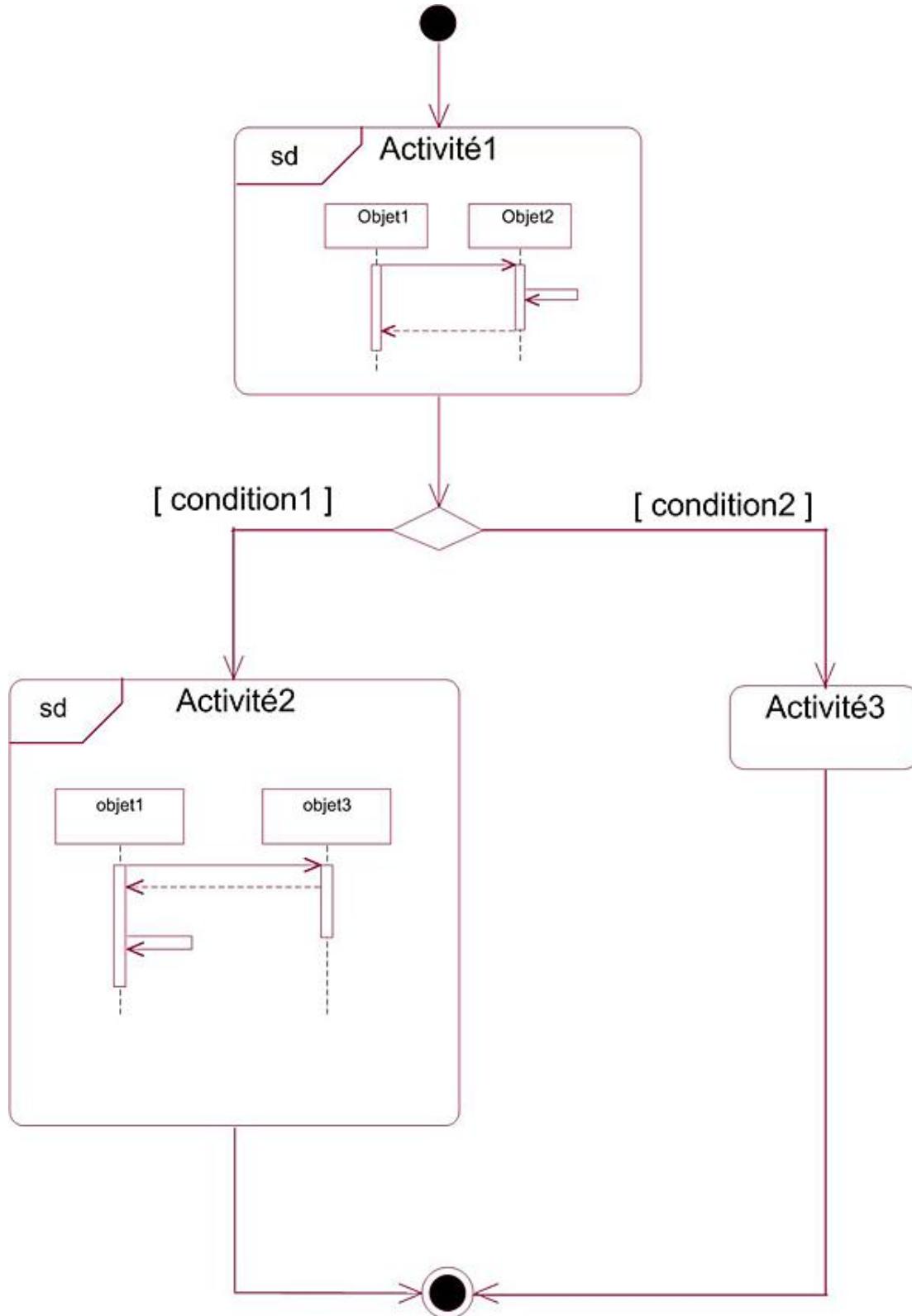


Figure 9.15 - Diagramme de vue d'ensemble des interactions

Conclusion

Le diagramme d'activités représente les activités que réalisent un ou plusieurs objets. Il peut correspondre à la description en détail d'une activité du diagramme d'états-transitions, à la description d'une méthode. Il peut également décrire l'activité d'un système ou d'un sous-système en assignant les responsabilités à chaque acteur. Le diagramme d'activités constitue aussi un bon choix pour décrire un cas d'utilisation.

Exercices

1. Le spectacle équestre

Construire le diagramme d'activités de l'achat d'un billet de spectacle équestre.

2. Le tiercé

Construire le diagramme d'activités de la vérification de la caisse d'un guichet de tiercé (uniquement par rapport aux billets vendus, sans prendre en compte le paiement des gains).

Introduction

Dans ce chapitre, nous aborderons les possibilités offertes par UML pour modéliser l'architecture du système. Cette modélisation se décline sous deux aspects :

- la modélisation de l'architecture logicielle et sa structuration en composants ;
- la modélisation de l'architecture matérielle et la répartition physique des logiciels.

Nous étudierons la notion de composant logiciel. Un composant est une boîte noire qui offre des services logiciels. Ces services sont décrits par une ou plusieurs interfaces du composant.

Nous avons étudié au chapitre La modélisation des objets la notion d'interface. Celle-ci s'applique également aux composants.

Rappelons qu'une interface est une classe abstraite ne contenant que des signatures de méthodes. La signature d'une méthode est composée de son nom et de ses paramètres.

Un composant peut également dépendre d'autres composants pour réaliser les services qu'il offre. Cette dépendance s'exprime sous la forme d'une interface requise qui décrit les services attendus.

La modélisation des composants et de leurs relations est décrite par le diagramme des composants.

La modélisation de l'architecture matérielle décrit les nœuds et leurs liaisons. Elle inclut la localisation des éléments logiciels au sein des nœuds, sous leur forme physique appelée *artefact*. Sa description est donnée par le diagramme de déploiement.

Le diagramme des composants

1. 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 langages de programmation non *objets* 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 composant :

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

➤ En UML 1, la notion de composant était large : un fichier exécutable, une bibliothèque partagée, un script étaient considérés comme des composants. En UML 2, la définition est réduite aux composants offrant des services au travers d'une ou de plusieurs interfaces fournies.

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 ses **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 le logo du composant. La figure 10.1 illustre la représentation graphique d'un composant sous deux formes, l'une avec le stéréotype, l'autre avec le logo.

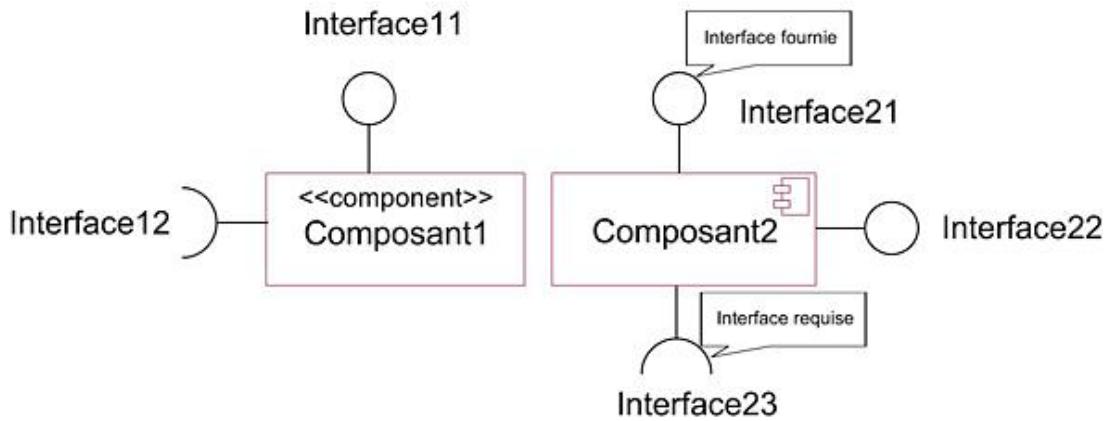


Figure 10.1 - Représentation graphique d'un composant et de ses interfaces

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 10.2 illustre cette représentation alternative. Celle-ci présente l'avantage de détailler les signatures de méthode contenues dans les interfaces.

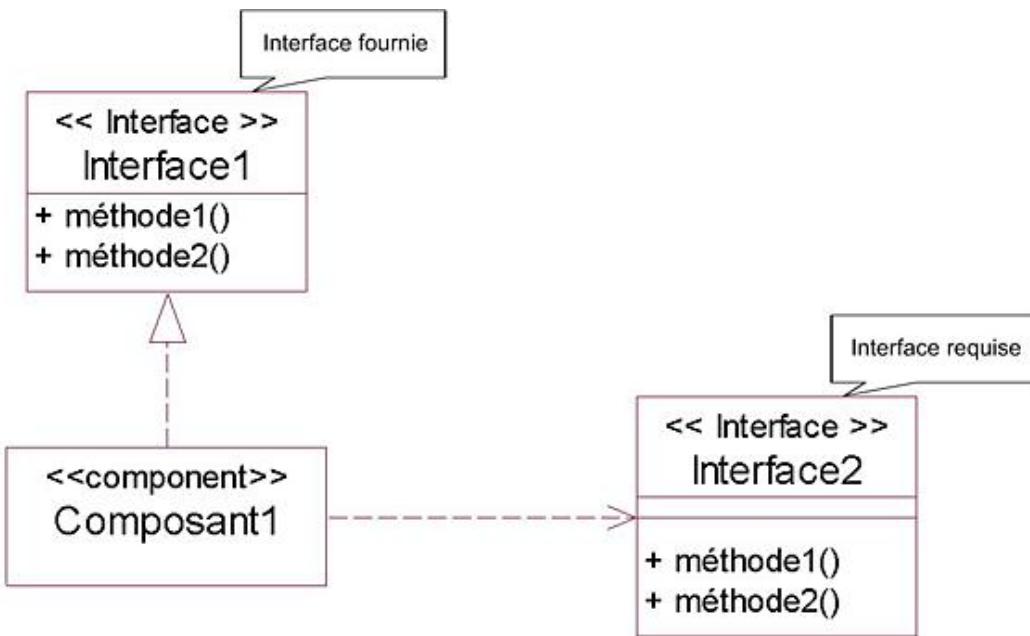


Figure 10.2 - Représentation alternative des interfaces

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 10.3 illustre ce composant.

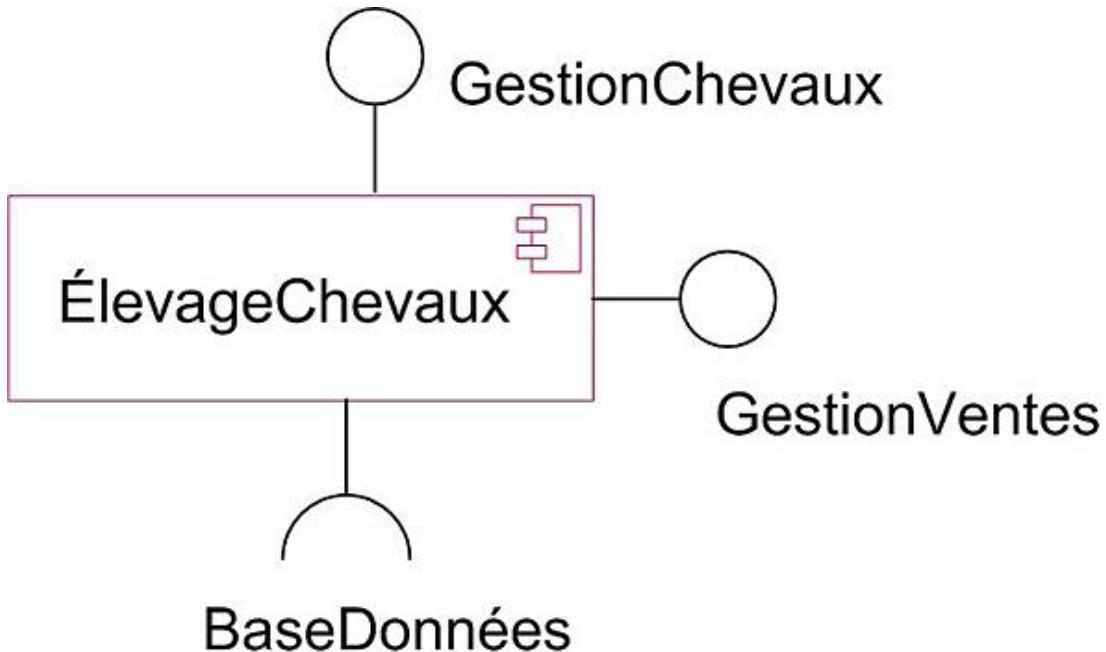


Figure 10.3 - Exemple de composant

2. L'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 des composants décrit cette architecture.

Exemple

Le système d'information d'un élevage de chevaux est réalisé par assemblage de composants. La figure 10.4 illustre le diagramme des composants de ce système. Les composants FenêtreGestionChevaux et FenêtreGestionVentes gèrent respectivement à l'écran une fenêtre consacrée à la gestion des chevaux et à la gestion des ventes de chevaux. Le composant

SystèmeBaseDonnées est un système de base de données.

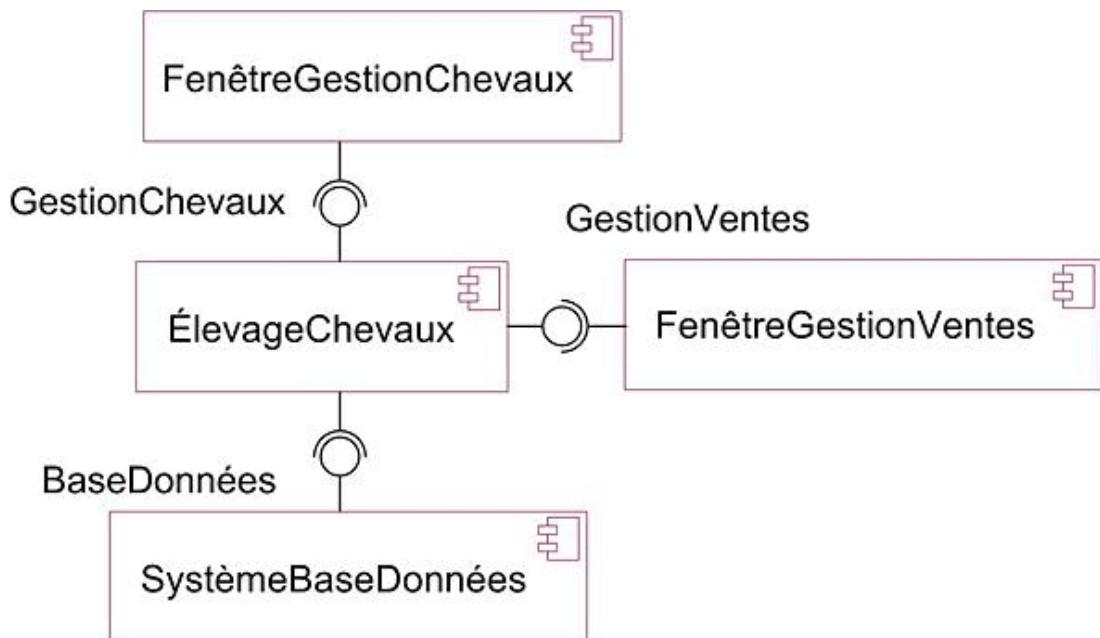


Figure 10.4 - Exemple de diagramme des composants

Le 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 du logiciel. 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 logiciels sous leur forme physique. Cette dernière est nommée artefact. Un fichier exécutable, une bibliothèque partagée ou un script sont des exemples de forme physique de logiciel.

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 10.5.

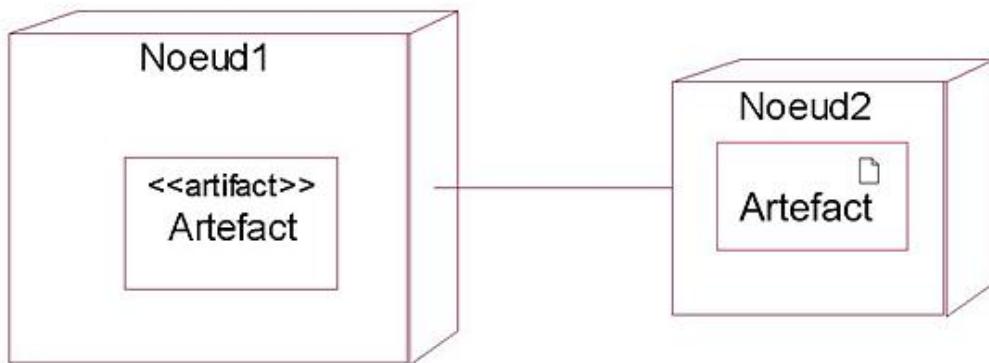


Figure 10.5 - Représentation graphique des nœuds, de leurs liens et des artefacts

Exemple

La figure 10.6 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.

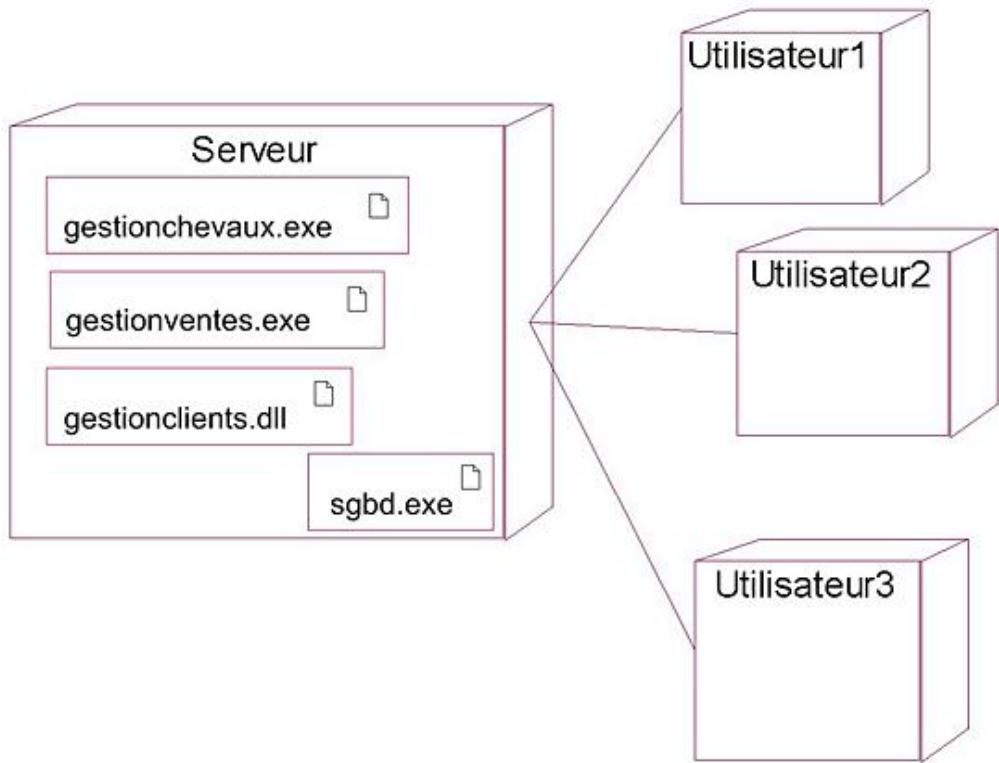


Figure 10.6 - Exemple de diagramme de déploiement

Conclusion

Le diagramme des composants et le diagramme de déploiement possèdent moins d'éléments que les diagrammes étudiés aux chapitres précédents. Ils sont cependant utiles lors de l'assemblage et du déploiement du système.

Introduction

Dans le cadre de MDA, nous présentons DB-MAIN. DB-MAIN est un outil CASE (*Computer Aided Software Engineering* ou Atelier de génie logiciel) orienté *objet* et destiné à la conception de systèmes d'information et plus précisément de bases de données relationnelles. DB-MAIN a été développé à l'université de Namur. Il est actuellement commercialisé par la société REVER de Charleroi.

DB-MAIN offre un processus de conception descendant : analyse des exigences avec la possibilité de décrire des cas d'utilisation et des diagrammes d'activité UML, conception du schéma à un niveau conceptuel, logique ou physique, intégration de schémas. Le niveau conceptuel de DB-MAIN correspond au modèle *objet* d'UML ou au modèle *Entité-Association* étendu à l'héritage. Le niveau logique correspond au modèle relationnel des données. Le niveau physique est celui du langage SQL propre aux SGBDR (*Système de gestion de base de données relationnel*). DB-MAIN inclut aussi d'autres modèles logiques et physiques, tels que IDS/2, IMS, fichiers standard ou XML.

Si le modélisateur choisit de travailler au niveau conceptuel, DB-MAIN réalise automatiquement la transformation vers le niveau logique (relationnel) puis vers le niveau physique (SQL). Cette transformation automatique rentre dans le cadre de l'approche MDA décrite au chapitre 2. Rappelons que MDA prône la réalisation de systèmes en faisant abstraction de la plateforme physique et de ses aspects technologiques. MDA introduit le PIM, *Platform Independent Model* ou modèle indépendant de la plate-forme, et le PSM, *Platform Specific Model* qui est le modèle spécifique à la plate-forme. Le passage de PIM à PSM doit se faire de façon automatisée ou semi-automatisée.

Dans le cadre de notre ouvrage, nous aborderons surtout cet aspect MDA de DB-MAIN, en retenant comme PIM, le modèle *objet* d'UML et comme PSM le modèle relationnel. Nous étudierons comment DB-MAIN transforme les classes, les associations et les relations d'héritage.

➤ DB-MAIN présente d'autres caractéristiques comme la rétroconception ou reverse engineering (analyse d'un schéma au niveau physique), la migration, l'intégration de bases de données ou la prise en compte de XML. Nous vous invitons à vous référer au site de DB-MAIN dont l'adresse est : www.db-main.be.

La transformation du modèle objet vers le modèle relationnel

1. La transformation des classes

Les classes deviennent des tables (encore appelées relations) dans le schéma relationnel.

Dans DB-MAIN, il est possible de spécifier qu'un ou plusieurs attributs d'une classe constituent une clef primaire, c'est-à-dire un identifiant unique des instances. Deux instances distinctes de la classe ne peuvent alors présenter les mêmes valeurs pour cet attribut ou cet ensemble d'attributs. Lors de la transformation, ceux-ci constituent alors la clef primaire de la table générée.

-
- Les méthodes ne sont pas prises en compte car il s'agit d'une transformation vers un PSM gérant uniquement des données.
-

La figure A.1 illustre un diagramme de classes dans DB-MAIN. Les attributs formant la clef primaire sont soulignés. La figure A.2 montre le diagramme transformé en schéma relationnel où les clefs sont montrées avec le préfixe *id*. Quant au préfixe *acc*, il signifie que les clefs primaires servent d'accès aux lignes de la table.

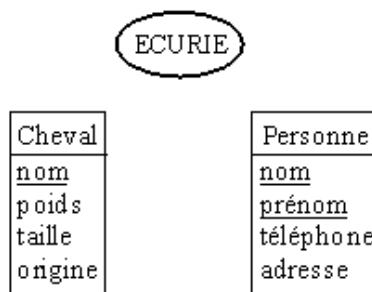


Figure A.1 - Diagramme de classes UML dans DB-MAIN

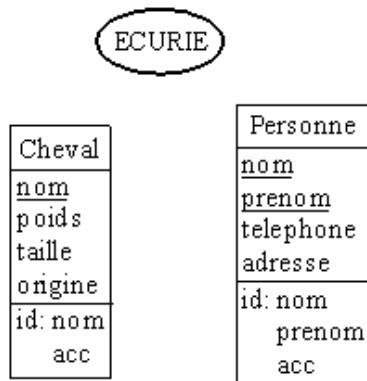


Figure A.2 - Transformation en schéma relationnel

-
- Le terme d'instance est réservé aux classes. Pour une table, le terme utilisé est ligne (raw en anglais) ou encore, dans les exposés théoriques, n-uplet (tuple en anglais).
-

-
- DB-MAIN offre également la possibilité d'introduire des clefs secondaires au niveau des classes. Ces clefs secondaires deviennent des clefs secondaires de la table générée.
-

2. La transformation des associations

a. Notion de clef étrangère

Dans le modèle relationnel, les clefs primaires sont utilisées comme support pour construire les associations. Pour qu'une ligne d'une table A puisse référencer une ligne d'une table B, une clef étrangère est introduite dans la table A. Cette clef étrangère est constituée d'attributs prenant les mêmes valeurs que les attributs de la clef primaire de la table B. La valeur de la clef étrangère doit correspondre à l'une des valeurs de la clef primaire pour l'une des lignes de la table B.

b. Associations dont une extrémité a pour cardinalité 0..1 ou 1..1

Les associations dont une extrémité a pour cardinalité 0..1 ou 1..1 sont traduites par l'introduction d'une clef étrangère dans la table située à l'autre extrémité.

La figure A.3 illustre cette transformation. Le préfixe *ref* sert à spécifier les clefs étrangères.

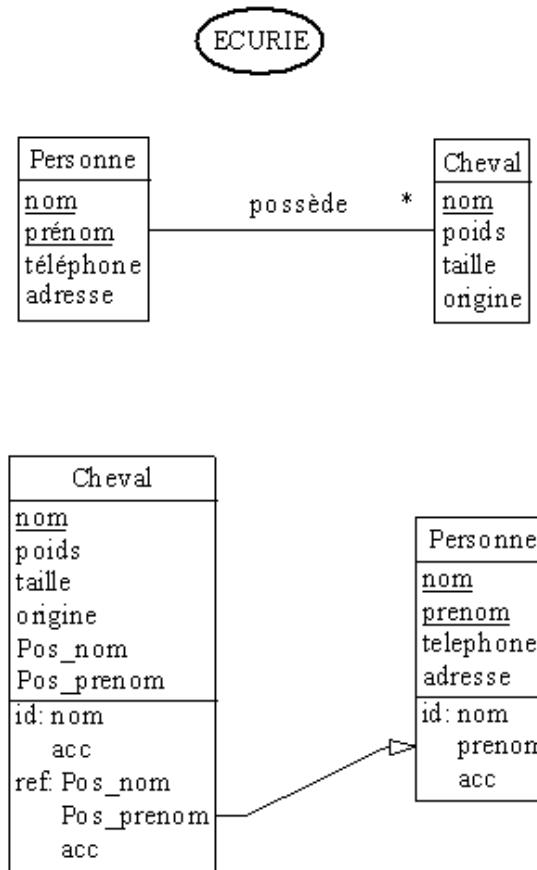


Figure A.3 - Transformation d'une association avec une cardinalité de un

c. Autres associations

Les autres associations (pour lesquelles la cardinalité maximale aux deux extrémités est supérieure à un) nécessitent la création d'une table supplémentaire pour être transformées dans le schéma relationnel.

Celle-ci est constituée de deux clefs étrangères correspondant chacune à la clef primaire des tables situées aux extrémités.

La figure A.4 illustre cette transformation sur un exemple. La table supplémentaire *monte* contient les mêmes attributs que les clefs primaires des deux tables situées aux extrémités de l'association. Ceux-ci servent à constituer les clefs étrangères de la table supplémentaire. Ces clefs étrangères forment également l'identifiant de la table supplémentaire.

Deux attributs étant désignés comme *nom*, DB-MAIN a automatiquement ajouté un préfixe à l'attribut introduit dans la classe *Cheval*.

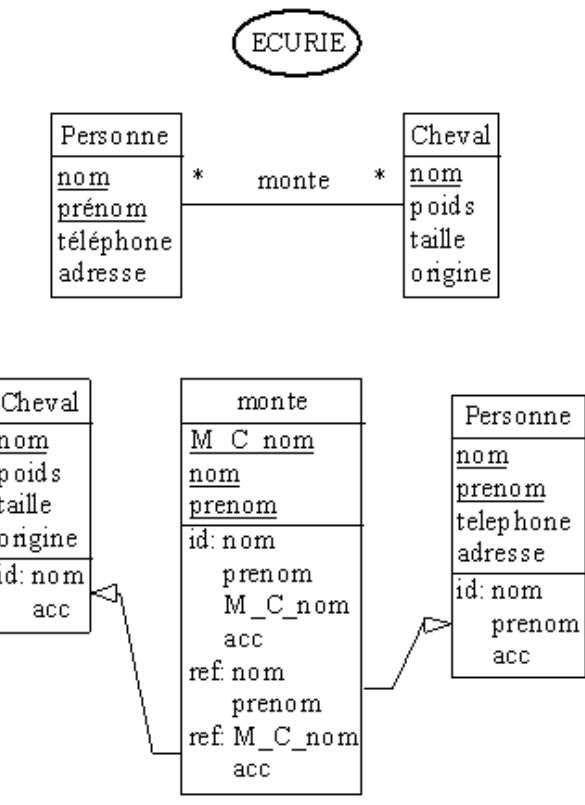


Figure A.4 - Transformation d'une association dont les cardinalités sont multiples

3. La transformation de l'héritage

a. Mécanisme de transformation

La relation d'héritage est transformée en une association dont la clef primaire se situe au niveau de la table correspondant à la surclasse et les clefs étrangères correspondantes au niveau des tables correspondant aux sous-classes.

La figure A.5 illustre cette transformation de la relation d'héritage sur un exemple.

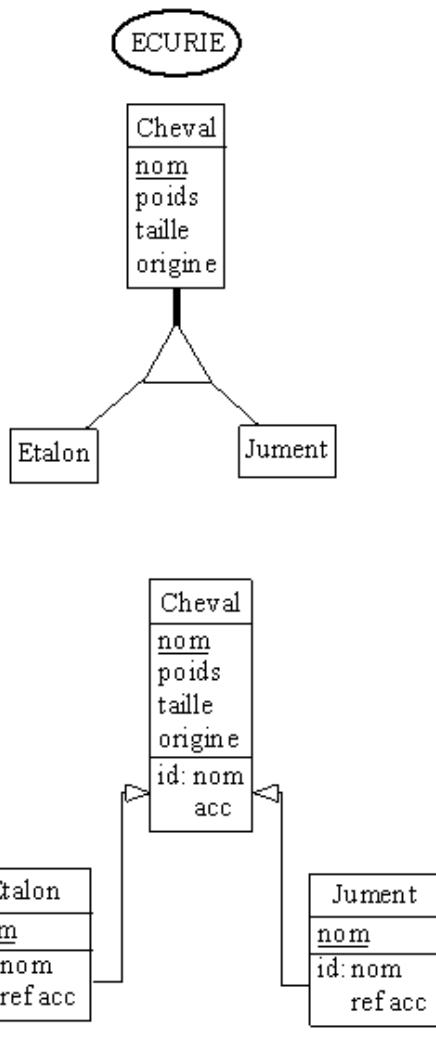


Figure A.5 - Transformation de la relation d'héritage

Une instance d'une sous-classe donne lieu à une ligne de la table issue de cette sous-classe. Cette ligne référence la ligne qui lui correspond dans la table de sa surclasse. Une table propose autant de clés étrangères de ce type qu'elle possède de surclasses. Pour retrouver l'ensemble des valeurs de l'instance, il faut procéder à une opération de jointure. De ce fait, cette transformation n'est pas très pratique dans le cas de hiérarchies complexes.

b. Prise en compte des contraintes liées à la relation d'héritage

Nous avons vu au chapitre La modélisation des objets qu'il existait quatre contraintes pouvant être exprimées au niveau des sous-classes :

- La contrainte {incomplete} signifie que l'ensemble des sous-classes est incomplet et qu'il ne couvre pas 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.

Ces quatre contraintes offrent quatre possibilités différentes détaillées dans le tableau suivant. On y trouve également le nom et le symbole de la contrainte correspondante dans DB-MAIN. Ce symbole apparaît au sein du triangle qui représente la relation d'héritage.

{incomplete} {overlapping}	pas de contrainte dans DB-MAIN	absence de symbole
{incomplete} {disjoint}	disjonction	symbole : D
{complete} {overlapping}	couverture	symbole : C
{complete} {disjoint}	partition	symbole : P

Pour gérer ces contraintes, DB-MAIN introduit des attributs servant de liens depuis la surclasse vers les sous-classes. Puis il ajoute une contrainte au niveau de ces liens pour exprimer la contrainte entre les sous-classes :

- La disjonction entre sous-classes est exprimée par la contrainte relationnelle *excl* : au plus un des attributs servant de liens prend une valeur.
- La couverture de la surclasse par ses sous-classes est exprimée par la contrainte relationnelle *at-lst-1* : au moins un des attributs servant de liens prend une valeur.
- La partition de la surclasse par ses sous-classes est exprimée par la contrainte relationnelle *exact-1* : exactement un des attributs servant de liens prend une valeur.

Un exemple de partition est représenté à la figure A.6. Les sous-classes *Étalon* et *Jument* constituent une partition de la surclasse *Cheval*. Un et un seul des attributs *Étalon* et *Jument* présents dans la classe *Cheval* prend une valeur. Ainsi toute instance de *Cheval* est obligatoirement une instance de *Jument* ou une instance de *Étalon*.

Au niveau SQL, le générateur standard produit du code qui exige du programmeur d'application une gestion explicite de ces attributs en fonction de la configuration des sous-classes. Le générateur paramétrique (pour Oracle jusqu'à présent) produit les composants (views, triggers, check) qui prennent complètement en charge ces contraintes ainsi que les opérations de mutation (c'est-à-dire de changement de classe d'une instance).

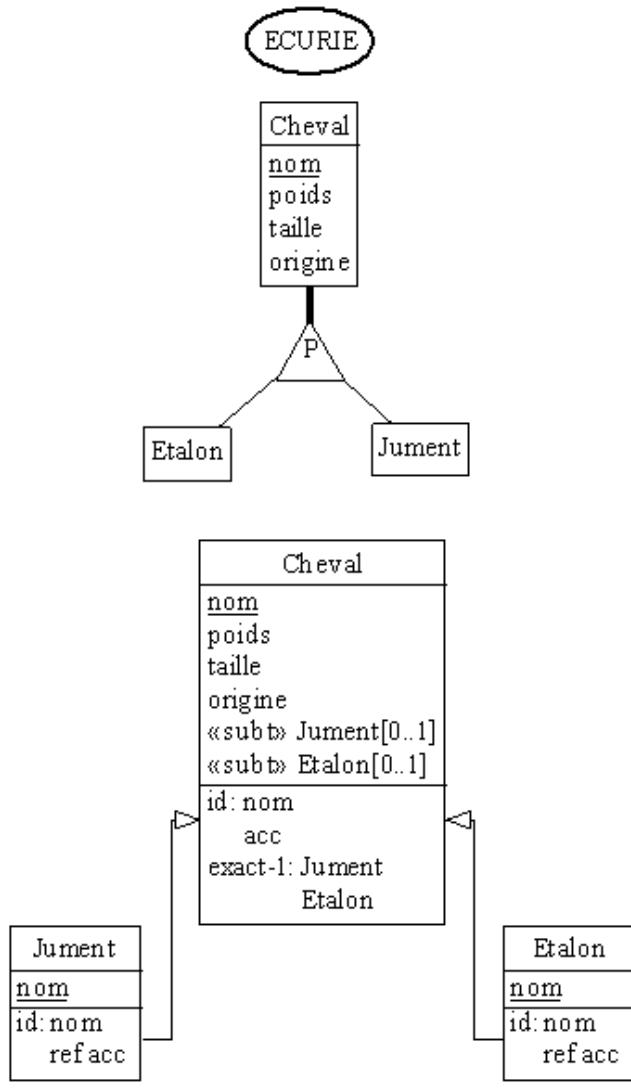


Figure A.6 - Transformation d'une partition

4. Conclusion

DB-MAIN est un outil CASE qui s'inscrit dans le cadre de l'approche MDA. Le PIM peut être le modèle *objet* d'UML ou le modèle *Entité-Association* étendu à l'héritage. Le PSM est le modèle relationnel. Ce dernier peut être automatiquement transformé en SQL. Il existe d'autres PSM tels que les fichiers standards ou les schémas XML. DB-MAIN présente l'avantage de réaliser cette transformation de façon approfondie, notamment en prenant en compte les contraintes liées à l'héritage. Notons également que DB-MAIN offre un passage immédiat du modèle *objet* d'UML au modèle *Entité-Association* et vice-versa.

Comme indiqué au chapitre À propos d'UML, le but de MDA est d'offrir une transformation automatique du PIM en PSM. Rappelons les principaux avantages de MDA :

- La conception est réalisée à un niveau plus abstrait, ce qui permet de se consacrer exclusivement à la spécification sans se préoccuper des contraintes du codage.
- Le fait de se consacrer uniquement à la réalisation du PIM apporte un vrai gain de productivité.
- Les aspects sémantiques sont spécifiés de façon plus explicite que s'ils étaient noyés dans du code, ce qui assure une meilleure lisibilité de la sémantique elle-même.
- Toute la sémantique doit être spécifiée au niveau du PIM pour que le PSM soit valide. Par conséquent, le PIM est nécessairement précis et rigoureux.

- La génération automatique du PSM depuis le PIM réduit fortement les besoins de rétroconception et apporte la portabilité vis-à-vis de la cible.
- Le problème de la mise à jour de la documentation du modèle lors des modifications au niveau du code est résolu. En effet, la documentation fonctionnelle est, en grande partie, constituée par le PIM.
- L'obligation de la conception du PIM impose de réaliser la phase de modélisation. Habituellement, cette phase n'est pas toujours prise en compte dans le cadre industriel qui veut donner priorité à la production, c'est-à-dire à la production du code. MDA confère à la modélisation un véritable apport en termes de gains de productivité.

Chapitre 4

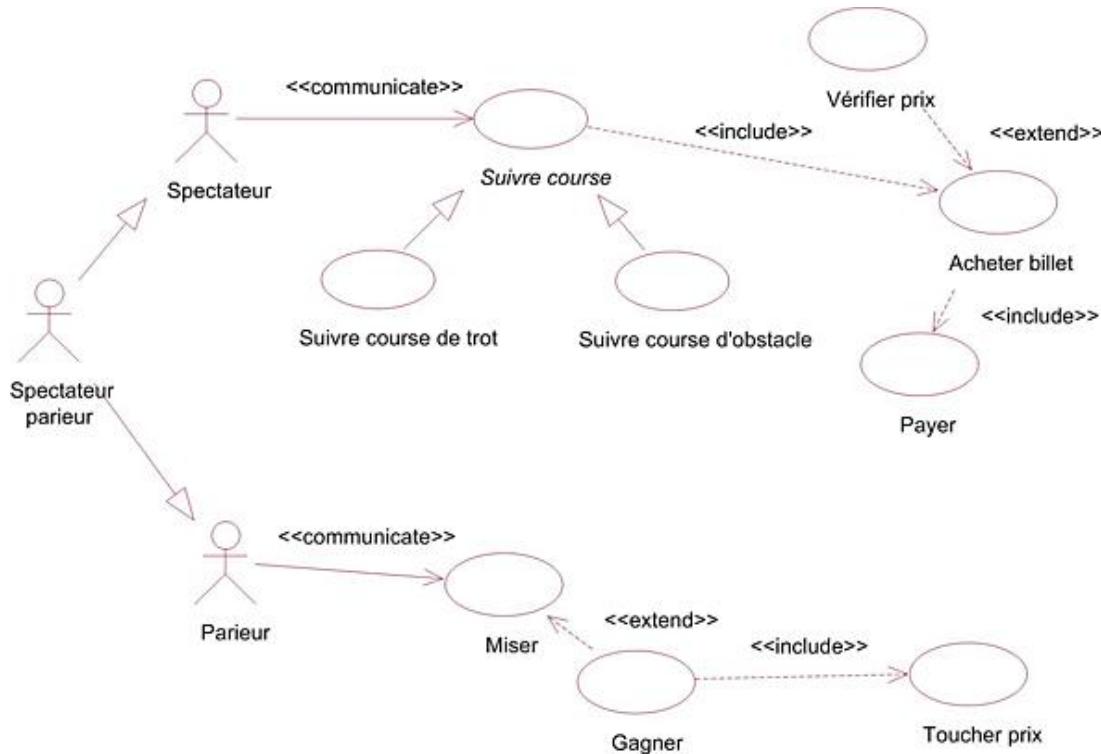
1. L'hippodrome

Un hippodrome offre à ses clients la possibilité de suivre les courses et de parier.

Quels sont les acteurs qui interagissent avec ces services ?

Le spectateur, le parieur et le client qui est à la fois spectateur et parieur.

Construire le diagramme des cas d'utilisation.



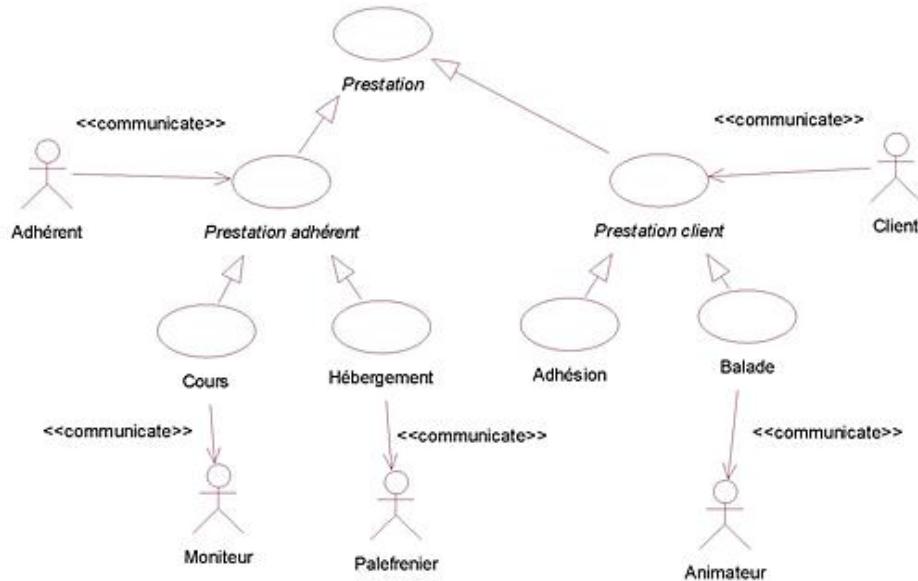
2. Le club équestre

Un club équestre offre les prestations d'hébergement des chevaux, de cours d'équitation, de balades. Seuls les adhérents ont accès aux cours et aux hébergements. Les autres clients ont la possibilité de faire des balades et d'adhérer.

Quels sont les acteurs qui interagissent avec ces services ?

Le moniteur, le palefrenier, l'animateur, l'adhérent et le client. L'adhérent et le client sont des acteurs primaires. Le moniteur, le palefrenier et l'animateur sont des acteurs secondaires.

Construire le diagramme des cas d'utilisation.

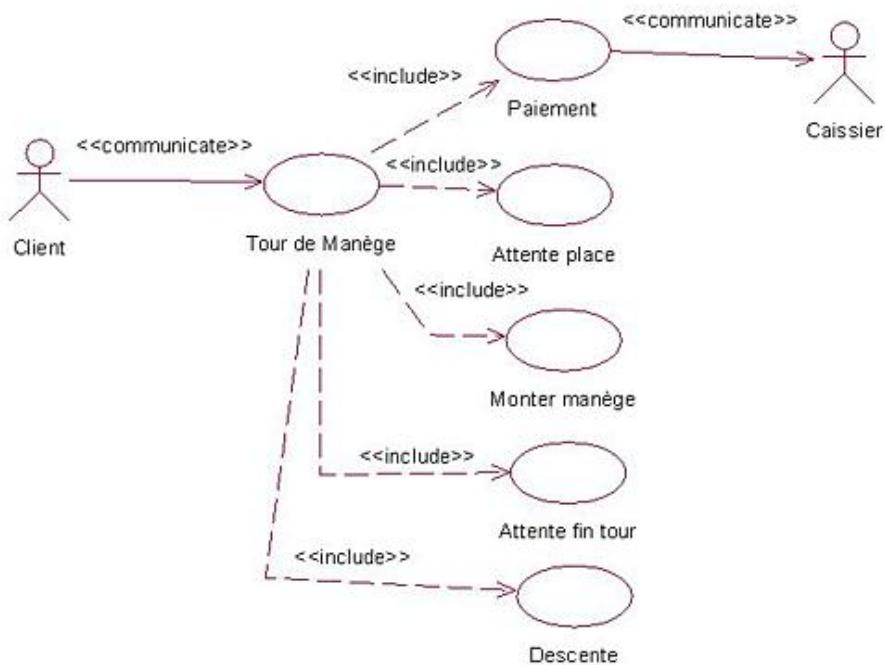


3. Le manège de chevaux de bois

Un manège de chevaux de bois offre à ses clients la possibilité de faire un tour de manège moyennant paiement.
Quels sont les acteurs liés à ce service ?

Le client et le caissier. Le client est un acteur primaire. Le caissier est un acteur secondaire.

Construire le diagramme des cas d'utilisation.



Donner la représentation textuelle correspondant au diagramme.

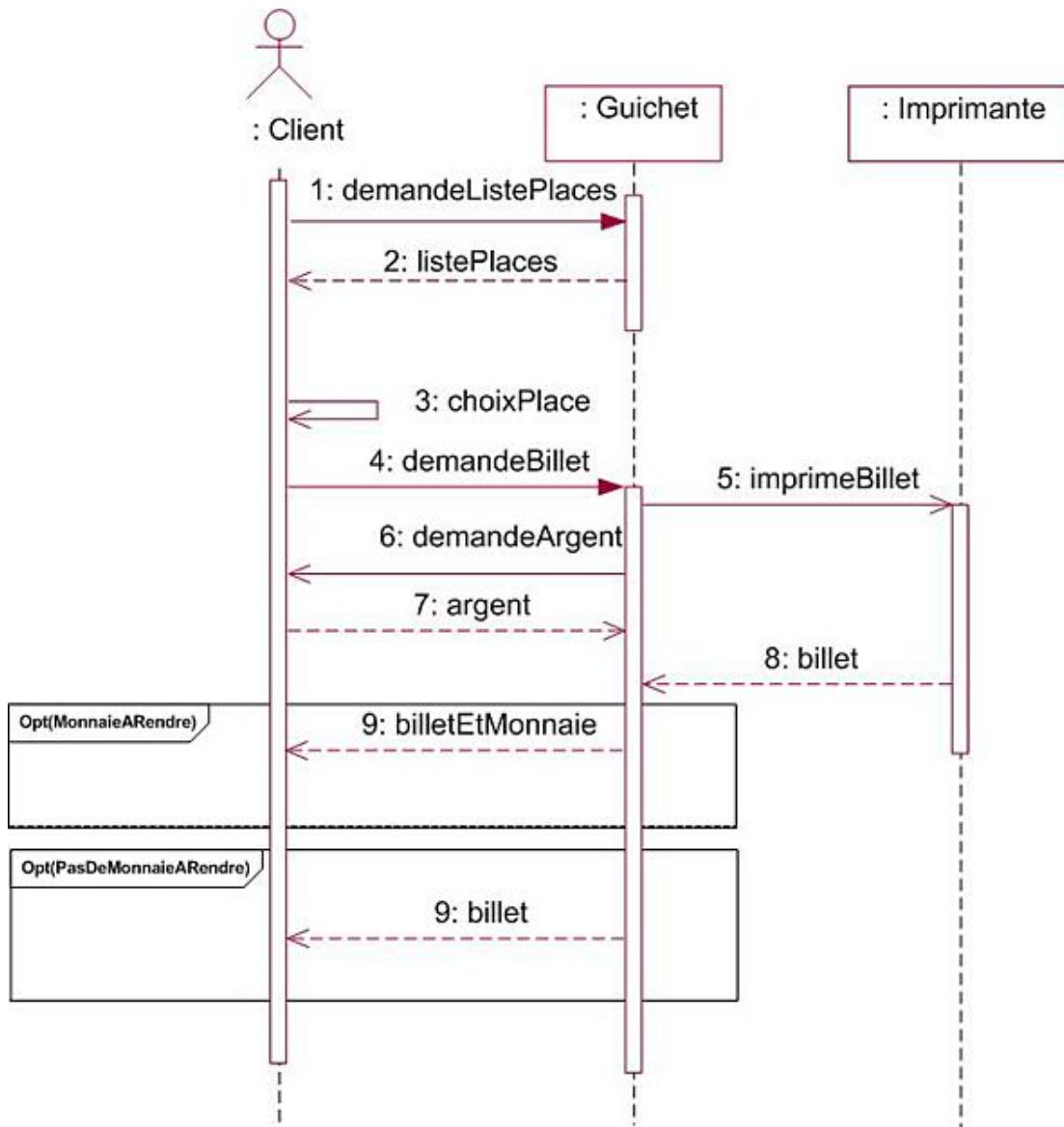
Cas d'utilisation	Tour de manège
-------------------	----------------

Acteur primaire	Client
Système	Manège de chevaux de bois
Intervenants	Client, Caissier
Niveau	Objectif utilisateur
Précondition	Le manège fonctionne
Opérations	
1	Paiement
2	Attendre une place
3	Monter sur le manège
4	Attendre la fin du tour
5	Descendre
Extensions	
1.A	Le client a-t-il assez d'argent ?
1.A.1	Si oui, continuer
1.A.2	Si non, abandonner
2.A	L'attente est-elle trop longue ?
2.A.1	Si oui, continuer
2.A.2	Si non, abandonner

Chapitre 5

1. L'hippodrome

Construire le diagramme de séquence d'achat d'un billet pour une course de chevaux.

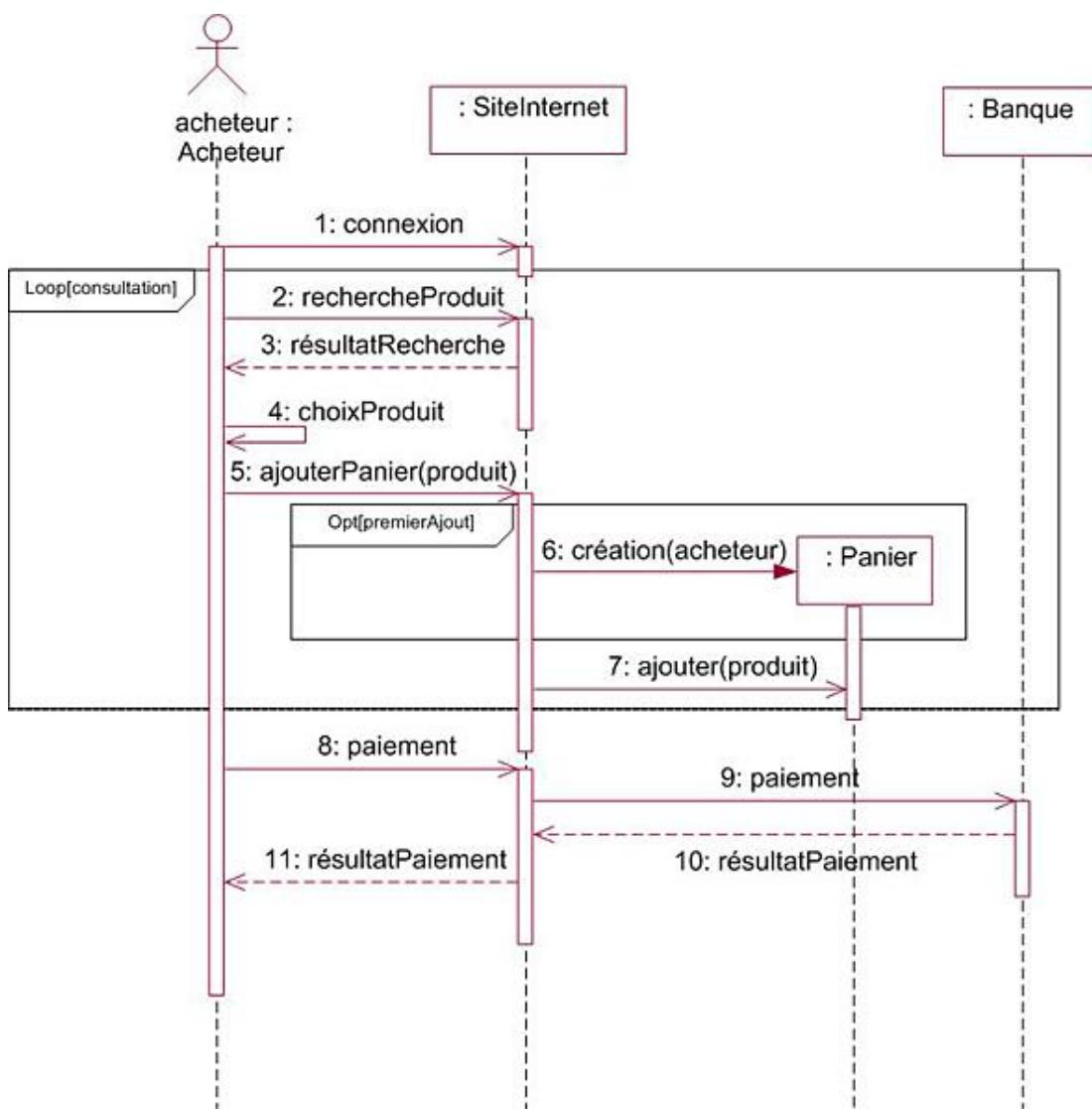


Quels sont les objets du système ainsi découverts ?

Le guichet et l'imprimante.

2. La centrale d'achat des chevaux

Construire le diagramme de séquence d'une prise de commande de produits sur le site Internet de la centrale d'achat des chevaux.



➤ Le diagramme de séquence n'inclut pas la livraison. Celle-ci peut faire l'objet d'un autre diagramme de séquence.

Quels sont les objets du système ainsi découverts ?

Le site Internet et le panier. La banque est un acteur secondaire.

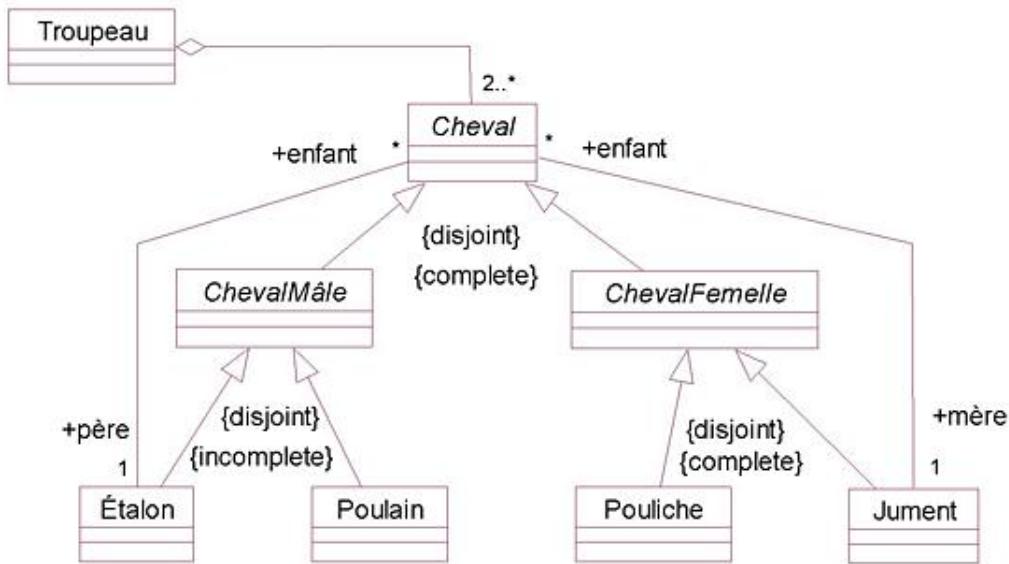
Chapitre 6

1. La hiérarchie des chevaux

Soit les classes *Jument*, *Étalon*, *Poulain*, *Pouliche*, *Cheval*, *Cheval mâle* et *Cheval femelle* ainsi que les associations père et mère. Établir la hiérarchie des classes en y faisant figurer les deux associations.

Utiliser les contraintes {incomplete}, {complete}, {disjoint} et {overlapping}.

Introduire la classe *Troupeau*. Établir l'association de composition entre cette classe et les classes déjà introduites.



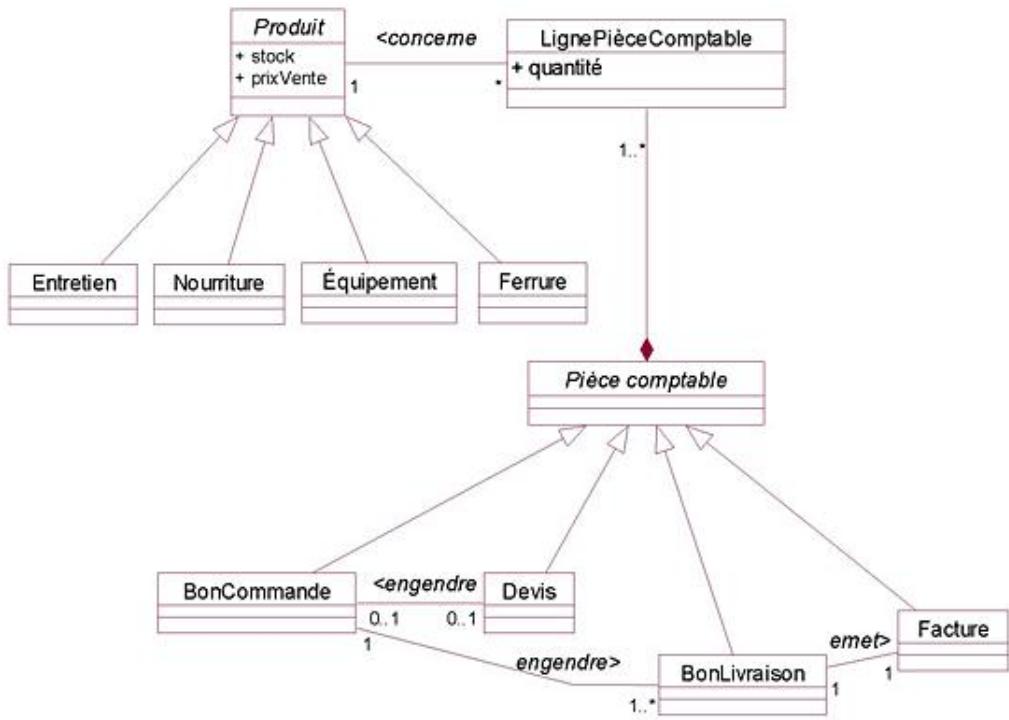
➤ Les classes *Étalon* et *Poulain* ne couvrent pas la classe *ChevalMâle* car il existe des chevaux châtrés.

2. Les produits pour chevaux

Modéliser les aspects statiques du texte suivant sous la forme d'un diagramme de classes.

Une centrale des chevaux vend différents types de produits pour chevaux : produits d'entretien, nourriture, équipement (pour monter le cheval), ferrures.

Une commande contient un ensemble de produits avec pour chacun d'eux, la quantité. Un devis est éventuellement établi avant le passage de la commande. En cas de rupture de stock, la commande peut engendrer plusieurs livraisons si le client le désire. Chaque livraison donne lieu à une facture.



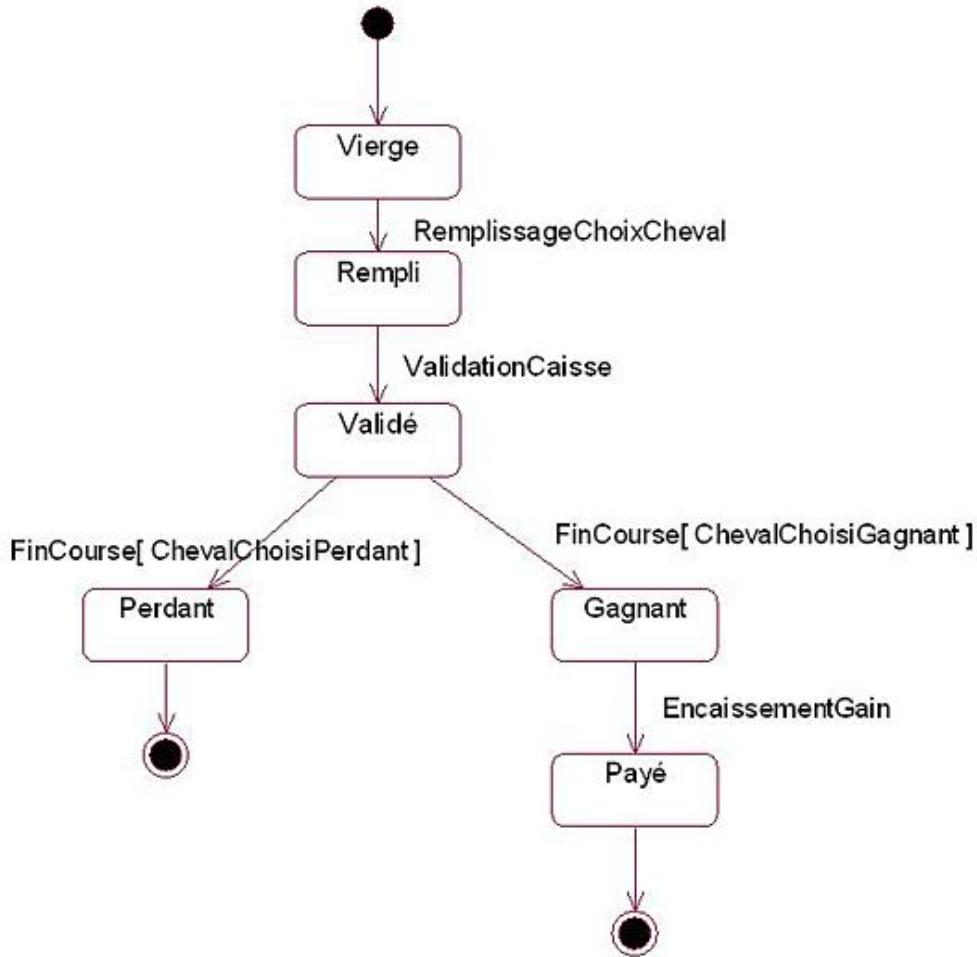
Chapitre 8

1. Le ticket de course de tiercé

Dans quels états peut se trouver un ticket de course de tiercé ?

Un ticket de course peut se trouver dans les états suivants : Vierge, Rempli, Validé, Perdant, Gagnant, Payé.

Construire le diagramme d'états-transitions d'une instance de la classe Ticket.

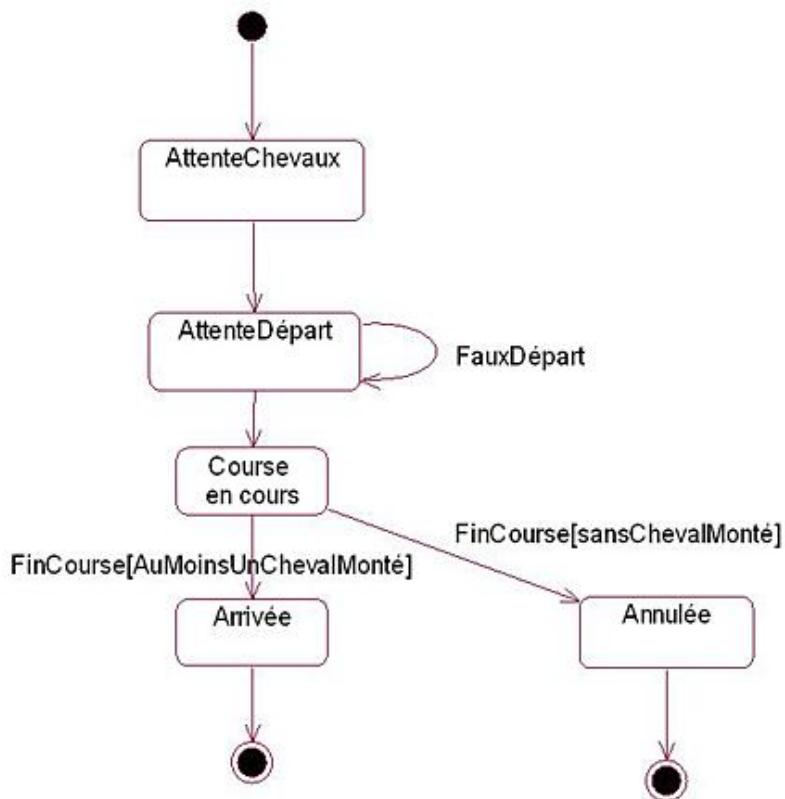


2. La course de chevaux

Dans quels états peut se trouver une course de chevaux ?

Une course de chevaux peut se trouver dans les états suivants : Attente des chevaux, Attente du départ, Course en cours, Arrivée, Annulée.

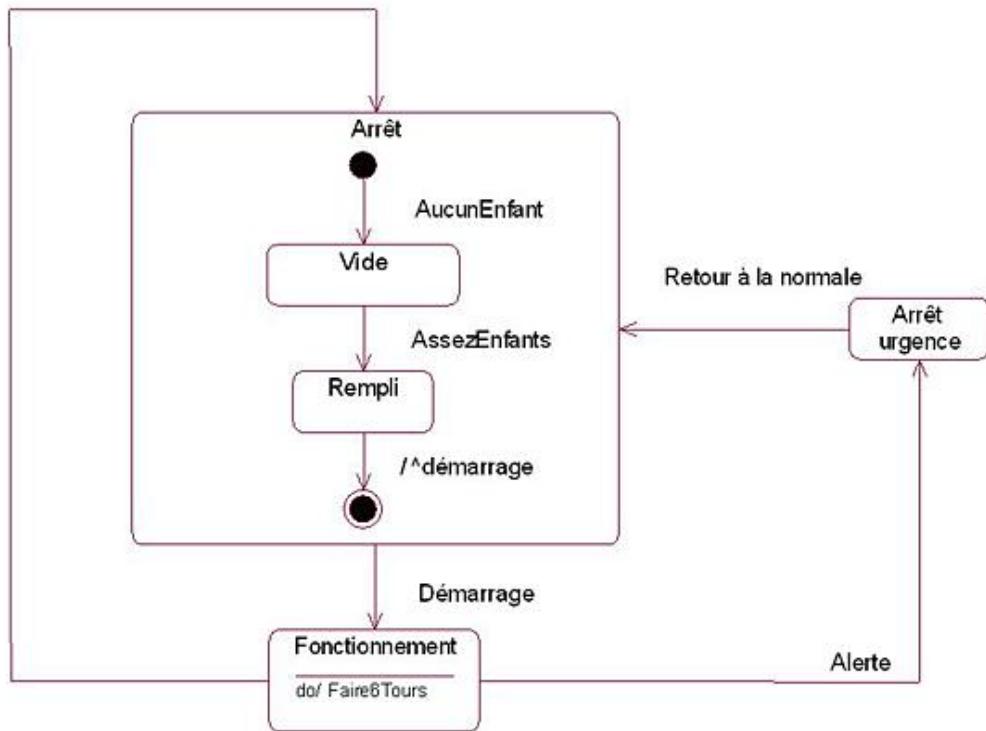
Construire le diagramme d'états-transitions d'une instance de la classe Course.



3. Le manège de bois

Décrire les différents états possibles d'un manège de chevaux de bois et construire le diagramme d'états-transitions correspondant.

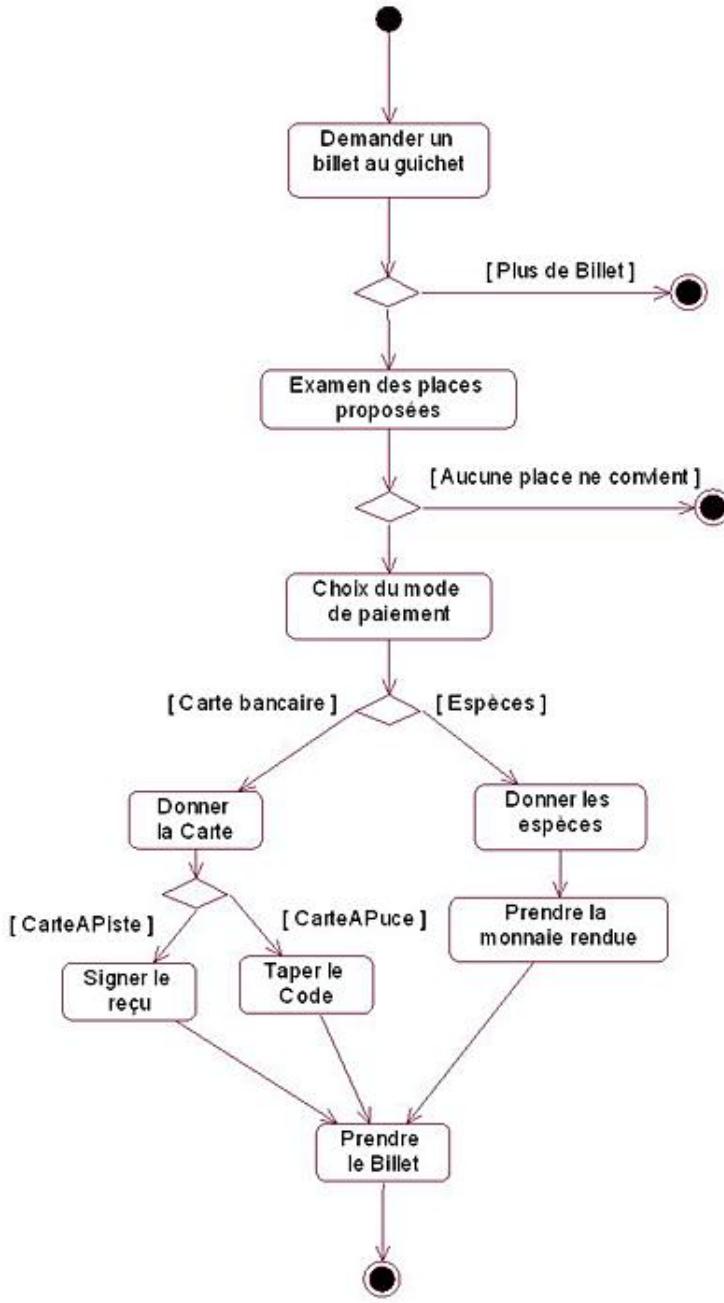
Les différents états possibles d'un manège de chevaux de bois sont les suivants : Il est à l'arrêt, en fonctionnement ou en arrêt d'urgence (après une alerte). Dans l'état d'arrêt, il peut être dans l'état rempli (avant un tour de manège) ou dans l'état vide (après un tour de manège quand tous les enfants ont quitté le manège).



Chapitre 9

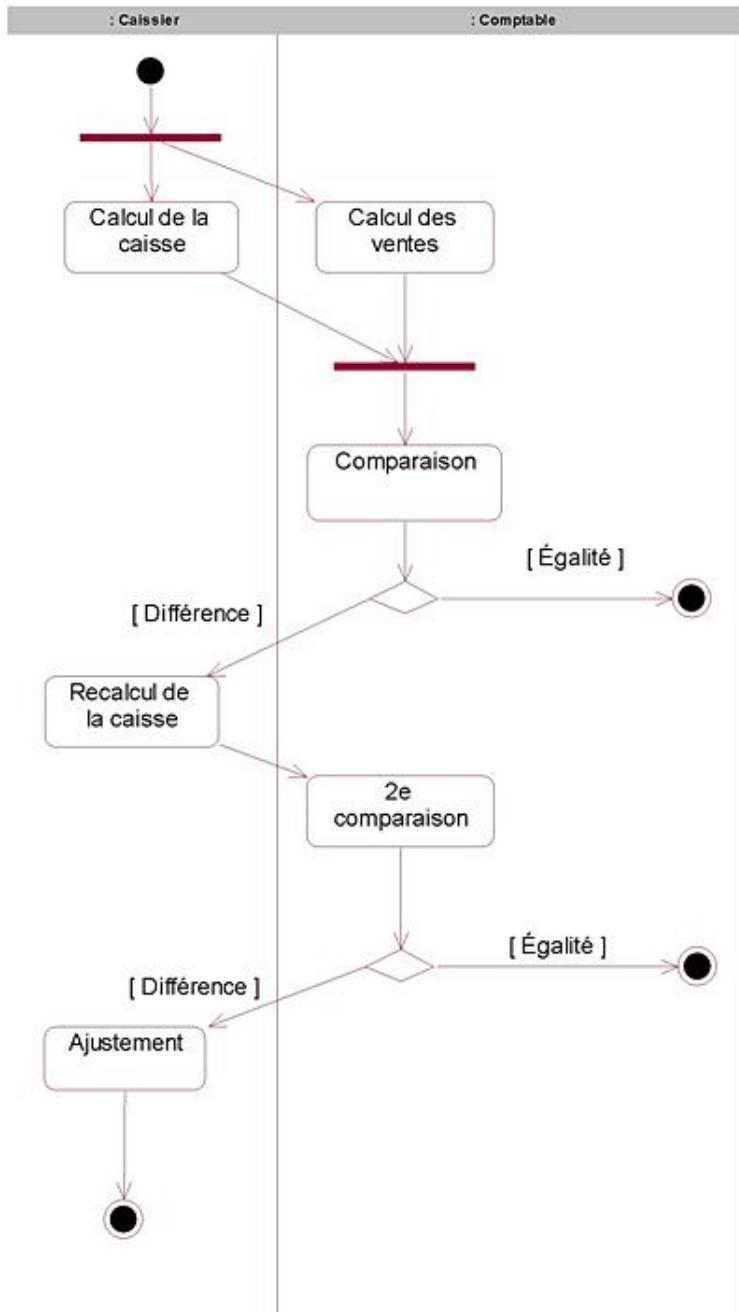
1. Le spectacle équestre

Construire le diagramme d'activités de l'achat d'un billet de spectacle équestre.



2. Tiersé

Construire le diagramme d'activités de la vérification de la caisse d'un guichet de tiersé (uniquement par rapport aux billets vendus sans prendre en compte le paiement des gains).



Glossaire

Acteur

Un acteur représente un utilisateur d'un cas d'utilisation dans son rôle vis-à-vis du système. Le nom de l'acteur est alors celui du rôle.

Deux catégories d'acteurs doivent être distinguées :

- les acteurs primaires pour lesquels l'objectif du cas d'utilisation est essentiel ;
- les acteurs secondaires qui interagissent avec le cas d'utilisation mais dont l'objectif n'est pas essentiel.

Activité

Une activité est une série d'actions. Une action consiste à affecter une valeur à un attribut, créer ou détruire un objet, effectuer une opération, envoyer un signal à un autre objet ou à soi-même, etc.

Activité composée

Le contenu d'une activité composée est formé d'autres activités.

Agrégation ou composition faible

L'agrégation est l'association qui relie un objet composé à ses composants. Elle est faible pour deux raisons : les composants peuvent appartenir à d'autres objets composés et la destruction de l'objet composé n'entraîne pas la destruction de ses composants.

Alternative

Dans un diagramme de séquence, l'alternative est l'un des opérateurs d'un cadre d'interaction. Elle est associée à une condition. Si la condition est vérifiée, le contenu du cadre est exécuté.

Dans un diagramme d'activités, l'alternative sert à sélectionner l'activité suivante. Chaque branche de l'alternative est dotée d'une condition de garde exclusive des autres conditions.

Artefact

Un artefact est constitué par la forme physique d'un logiciel. Un fichier exécutable, une bibliothèque partagée ou un script sont des exemples de forme physique de logiciel.

Association entre objets

Une association entre objets est un ensemble de liens entre les instances de deux ou plusieurs classes. Elle est décrite dans le diagramme des classes.

Associations entre paquetages

Il existe deux associations entre paquetages :

- L'association 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.
- L'association 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.

Association réflexive

Une association réflexive relie les instances d'une classe entre elles.

Attribut calculé

La valeur d'un attribut calculé est donnée par une fonction basée sur la valeur d'autres attributs.

Attribut de classe

Un attribut de classe est lié à la classe elle-même et non à chaque instance. Un tel attribut est partagé par l'ensemble des instances de la classe.

Boucle

Dans un diagramme de séquence, la boucle est l'un des opérateurs d'un cadre d'interaction. Elle consiste en une exécution répétée du contenu du cadre tant que la condition de fin n'est pas remplie ou que le nombre maximal de répétitions n'est pas atteint.

Cas d'utilisation

Un cas d'utilisation décrit les interactions, entre un utilisateur et le système.

Dans un cas d'utilisation avec objectif de l'utilisateur, cette suite d'interactions est liée à un objectif fonctionnel de l'utilisateur.

Dans un cas d'utilisation de sous-fonction, cette suite d'interactions est destinée à être incluse dans un autre cas d'utilisation.

Cadre d'interaction

Un cadre d'interaction est une partie du diagramme de séquence associée à une étiquette contenant un opérateur qui en détermine la modalité d'exécution. Les principales modalités sont le branchement conditionnel et la boucle.

Cardinalité minimale ou cardinalité maximale

Une cardinalité est fixée à une extrémité d'une association. Une cardinalité minimale, respectivement maximale, permet de fixer le nombre minimal, respectivement maximal, d'instances auxquelles une instance de la classe située à l'autre extrémité de l'association est reliée.

Classe

Une classe est constituée par un ensemble d'objets similaires possédant les mêmes attributs et méthodes. Cette représentation commune est définie en commun au niveau de la classe.

Une classe concrète définit un modèle complet. Elle possède des instances directes.

Une classe abstraite définit un modèle abstrait. Elle ne possède pas d'instances directes. Une telle classe sert à factoriser, en tant que surclasse, des attributs et des méthodes communes de plusieurs classes concrètes.

Une classe-association est à la fois une association et une classe dont les instances sont les occurrences de l'association. Ainsi, ces occurrences peuvent être dotées d'attributs ou d'opérations.

Composant

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.

Composition

La composition (ou composition forte) est l'association qui relie un objet à ses composants. Elle est forte pour deux raisons : les composants ne peuvent pas appartenir à d'autres objets composés et la destruction de l'objet composé entraîne la destruction de ses composants.

Condition de garde

Une condition de garde est utilisée dans les diagrammes de communication, d'états-transitions et d'activités. Elle constitue une condition pour respectivement envoyer le message, franchir la transition ou enchaîner les activités.

Contraintes sur la relation d'héritage

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

- {incomplete} : l'ensemble des sous-classes est incomplet et ne couvre pas la surclasse, c'est-à-dire que l'ensemble des instances des sous-classes est un sous-ensemble de l'ensemble des instances de la surclasse.
- {complete} : l'ensemble des sous-classes est complet et couvre la surclasse.
- {disjoint} : les sous-classes n'ont aucune instance en commun.
- {overlapping} : les sous-classes peuvent avoir une ou plusieurs instances en commun.

Couloir

Un couloir regroupe toutes les activités dont un même objet est le responsable.

Cycle de vie

Le cycle de vie d'un objet est l'ensemble de ses états et des transitions les reliant.

Diagramme d'activités

Ce diagramme décrit les activités d'un ou de plusieurs objets ainsi que leurs enchaînements.

Diagramme de cas d'utilisation

Ce diagramme décrit l'ensemble (ou un sous-ensemble) des cas d'utilisation et d'acteurs d'un système ainsi que les associations les reliant.

Diagramme de classes

Ce diagramme décrit l'ensemble (ou un sous-ensemble) des classes et interfaces d'un système ainsi que les associations qui les relient.

Diagramme de communication

Ce diagramme décrit les interactions entre un ensemble d'objets en montrant, de façon spatiale, les envois de message intervenant entre eux.

Diagramme des composants

Ce diagramme montre la structuration en composants logiciels d'un système.

Diagramme de déploiement

Ce diagramme décrit l'architecture matérielle d'un système.

Diagramme d'états-transitions

Ce diagramme illustre l'ensemble des états du cycle de vie d'un objet séparés par des transitions.

Diagramme des objets

Ce diagramme montre, à un moment donné, les instances créées et leurs liens lorsque le système est actif.

Diagramme de paquetage

Ce diagramme est un regroupement d'éléments de modélisation.

Diagramme de séquence

Ce diagramme décrit les interactions entre un ensemble d'objets en montrant, de façon séquentielle, les envois de message qui interviennent entre eux.

Diagramme de timing

Le diagramme de timing montre les changements d'état d'un objet en fonction du temps.

Diagramme de vue d'ensemble des interactions

Le diagramme de vue d'ensemble des interactions est un diagramme d'activités où chaque activité peut être décrite par un diagramme de séquence.

Encapsulation

L'encapsulation consiste à masquer la structure et le comportement internes et propres au fonctionnement de l'objet. Ce masquage peut être complet (encapsulation privée), ne pas s'appliquer aux sous-classes (encapsulation protégée) ou ne pas s'appliquer aux classes du même paquetage (encapsulation de paquetage).

Enchaînement d'activités

Un enchaînement d'activités est un lien depuis une activité d'origine vers une activité de destination. Il est franchi lorsque l'activité d'origine est terminée.

Envoi de message

Voir *Message*.

État

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 réaliser une activité ou attendre un signal provenant d'autres objets.

Généralisation

La généralisation est la relation qui lie une sous-classe à sa superset (ou l'une des supersedes en cas d'héritage multiple).

La généralisation s'applique également aux cas d'utilisation.

Granularité

La granularité d'un objet représente sa taille. Un objet de petite taille est dit de granularité fine ou de petit grain. Un objet volumineux est dit de granularité importante ou de gros grain.

Héritage

L'héritage est la propriété qui fait bénéficier une sous-classe de la structure et du comportement de sa superset.

L'héritage est multiple quand une sous-classe possède plusieurs supersedes.

Interface

Une interface est une classe abstraite ne contenant que des signatures de méthodes. La signature d'une méthode est composée de son nom et de ses paramètres.

Une interface fournie décrit les services offerts par un composant. Une interface requise décrit les services qu'un composant attend d'un autre composant dont il est le client.

Instance

Une instance d'une classe est un élément de l'ensemble des objets de cette classe.

Ligne de vie

Au sein d'un diagramme de séquence, une ligne de vie montre les actions et réactions d'une instance, ainsi que les périodes pendant lesquelles elle est active.

MDA

MDA (*Model Driven Architecture* ou architecture guidée par les modèles) est une proposition de l'OMG dont l'objectif est la conception de systèmes basée sur la seule modélisation du domaine, indépendamment de la plateforme.

Message

Un message est envoyé à un objet pour l'activer et provoquer l'exécution de la méthode de même nom. Un envoi de message est un appel de méthode.

Un message peut être envoyé de façon asynchrone. Dans ce cas, l'appelant atteint la fin de l'exécution de la méthode de l'objet récepteur avant de continuer son exécution.

Un message peut également être envoyé de façon synchrone. Dans ce cas, l'appelant continue son exécution immédiatement après l'envoi du message.

Méthode

Une méthode est un ensemble d'instructions prenant des valeurs en entrée et modifiant les valeurs des attributs ou produisant un résultat.

L'ensemble des méthodes d'une classe décrit le comportement des instances de cette classe.

Méthode de classe

Une méthode de classe est liée à la classe elle-même et non à une instance. Son invocation se fait au travers de la classe et non de l'une de ses instances.

Navigation

La navigation d'une association en détermine le sens de parcours.

Nœud

Un nœud est une unité matérielle capable de recevoir et d'exécuter du logiciel.

Objet

Un objet est une entité identifiable du monde réel. Dans le modèle UML, un objet est une instance d'une classe.

Occurrence d'une association

Une occurrence d'une association est un lien entre des instances des classes situées aux extrémités de cette association.

OCL

OCL (*Object Constraint Language* ou langage de contraintes objet) est un langage destiné à exprimer les contraintes au sein d'un diagramme de classes sous forme de conditions logiques.

OMG

L'OMG ou Object Management Group est un consortium formé de plus de 800 sociétés et universités qui a pour but de promouvoir les technologies de l'objet.

Paquetage

Un paquetage est un regroupement d'éléments de modélisation : classes, composants, cas d'utilisation, autres paquetages.

PIM

Le PIM (*Platform Independant Model* ou modèle indépendant de la plateforme) est le modèle de conception de l'architecture MDA.

Polymorphisme

Le polymorphisme est la différence de comportement qui existe entre des sous-classes d'une même superclasse pour les méthodes de même nom.

Processus

Un processus est un ensemble d'opérations prenant en entrée des données et produisant de nouvelles données.

Processus unifié

Le Processus Unifié est un processus de conception et d'évolution de logiciels basé sur UML.

PSM

Le PSM (*Platform Specific Model* ou modèle spécifique de la plate-forme) est le modèle cible de l'architecture MDA.

Qualification

Une association peut être qualifiée à une extrémité afin de réduire à l'autre extrémité la cardinalité maximale. En effet, la valeur du qualificateur est alors prise en compte pour déterminer le nombre de liens.

Relation de communication

La relation de communication lie un acteur à un cas d'utilisation.

Relation d'extension

La relation d'extension permet d'enrichir un cas d'utilisation par un cas d'utilisation de sous-fonction. Cet enrichissement est optionnel.

Relation d'inclusion

La relation d'inclusion permet d'enrichir un cas d'utilisation par un cas d'utilisation de sous-fonction. Cet enrichissement est obligatoire.

Relation de réalisation

La réalisation d'une interface, c'est-à-dire l'implantation de ses méthodes est confiée à une ou plusieurs classes concrètes, sous-classes de l'interface. La relation d'héritage qui existe entre une interface et une sous-classe d'implantation est appelée relation de réalisation.

Cette relation existe également entre une interface et un composant qui implante ses méthodes.

RUP

RUP (*Rational Unified Process* ou processus unifié de Rational) est la version sous forme de base documentaire du processus unifié.

Scénario

Un scénario est une instance d'un cas d'utilisation dont toutes les alternatives ont été fixées.

Spécialisation

La spécialisation est la relation qui lie une superset à l'une de ses sous-classes.

La spécialisation s'applique également aux cas d'utilisation.

Stéréotype

Un stéréotype est un mot clé utilisé pour expliciter la spécialisation d'un élément. Un stéréotype est noté entre guillemets.

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.

Type

Le type peut être une classe ou un type standard. Les types standard sont respectivement 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.

UML

UML (*Unified Modeling Language* ou langage unifié de modélisation) est un langage graphique destiné à la modélisation de

systèmes et de processus.

Français-anglais

Abstrait	abstract
Acteur	actor
Activité composée	composite activity
Activité	activity
Agrégation (ou composition faible)	aggregation (or weak composition)
Alternative	choice
Artefact	artifact
Association réflexive	reflexive association
Attribut calculé	derived attribute
Attribut de classe	class attribute
Booléen	boolean
Boucle	loop
Cadre d'interaction	interaction fragment
Cardinalité minimale, maximale	minimal, maximal multiplicity
Cas d'utilisation	use case
Chaîne de caractères	string
Classe	class
Colonne	column
Composant	component
Composition (forte)	(strong) composition
Condition de garde	guard condition
Contrainte sur la relation d'héritage	inheritance relationship constraint
Couloir	Swimlane
Cycle de vie	lifecycle
Dépendance	dependency
Diagramme d'activités	activity diagram
Diagramme d'états-transitions	statechart diagram or state diagram
Diagramme d'interaction	interaction diagram

Diagramme des objets	object diagram
Diagramme de cas d'utilisation	use case diagram
Diagramme de classes	class diagram
Diagramme de communication	communication diagram
Diagramme de déploiement	deployment diagram
Diagramme de paquetage	package diagram
Diagramme de séquence	sequence diagram
Diagramme de timing	timing diagram
Diagramme de vue d'ensemble des interactions	interaction overview diagram
Diagramme des composants	component diagram
Encapsulation	encapsulation
Enchaînement d'activités	activity edge
Entier	integer
Envoi de message	message sending
Etat	state
Faux	false
Généralisation	generalisation
Granularité	granularity
Héritage	inheritance
Instance	instance
Interface	interface
Langage de contraintes objet (OCL)	Object Constraint Language (OCL)
Langage unifié de modélisation (UML)	Unified Modeling Language (UML)
Lien (entre objets)	link (between objects)
Ligne	raw
Ligne de vie	lifeline
MDA (Architecture guidée par les modèles)	MDA (<i>Model Driven Architecture</i>)
Message	message
Méthode	method
Méthode de classe	class method

Modèle indépendant de la plate-forme (PIM)	Platform Independent Model (PIM)
Modèle spécifique à la plate-forme (PSM)	Platform Specific Model (PSM)
Navigation	navigation
Nœud	node
Objet	object
Paquetage	package
Polymorphisme	polymorphism
Processus	process
Processus Unifié	Unified Process (UP)
Qualificateur	qualifier
Qualification	qualification
Relation d'extension	extension relationship
Relation d'inclusion	inclusion relationship
Relation de communication	communication relationship
Relation de réalisation	realisation relationship
Scénario	scenario
Si	if
Sinon	else
Spécialisation	specialisation
Stéréotype	stereotype
Système	system
Transition	transition
Type	type
Vrai	true

Anglais-français

Abstract	abstrait
Activity diagram	diagramme d'activités
Activity edge	enchaînement d'activités
Activity	activité
Actor	acteur
Aggregation (or weak composition)	agrégation (ou composition faible)
Artifact	artefact
Boolean	booléen
Choice	alternative
Class attribute	attribut de classe
Class diagram	diagramme de classes
Class method	méthode de classe
Class	classe
Column	colonne
Communication diagram	diagramme de communication
Communication relationship	relation de communication
Component diagram	diagramme des composants
Component	composant
Composite activity	activité composée
Composition (strong composition)	composition (forte)
Dependency	dépendance
Deployment diagram	diagramme de déploiement
Derived attribute	attribut calculé
Else	sinon
Encapsulation	encapsulation
Extension relationship	relation d'extension
False	faux
Generalisation	généralisation

Granularity	granularité
Guard condition	condition de garde
If	si
Inclusion relationship	relation d'inclusion
Inheritance relationship constraint	contrainte sur la relation d'héritage
Inheritance	héritage
Instance	instance
Integer	entier
Interaction diagram	diagramme d'interaction
Interaction fragment	cadre d'interaction
Interaction overview diagram	diagramme de vue d'ensemble des interactions
Interface	interface
Lifecycle	cycle de vie
Lifeline	ligne de vie
Link (between objects)	lien (entre objets)
Loop	boucle
MDA (<i>Model Driven Architecture</i>)	MDA (Architecture guidée par les modèles)
Message sending	envoi de message
Message	message
Method	méthode
Multiplicity (minimal, maximal multiplicity)	cardinalité minimale, maximale
Navigation	navigation
Node	nœud
Object Constraint Language (OCL)	langage de contraintes objet (OCL)
Object diagram	diagramme des objets
Object	objet
Package	paquetage
Package diagram	diagramme de paquetage
Platform Independent Model (PIM)	modèle indépendant de la plate-forme (PIM)
Platform Specific Model (PSM)	modèle spécifique à la plate-forme (PSM)

Polymorphism	polymorphisme
Process	processus
Qualification	qualification
Qualifier	qualificateur
Raw	ligne
Realisation relationship	relation de réalisation
Reflexive association	association réflexive
Scenario	scénario
Sequence diagram	diagramme de séquence
Specialisation	spécialisation
State	état
Statechart diagram (or state diagram)	diagramme d'états-transitions
Stereotype	stéréotype
String	chaîne de caractères
Swimlane	couloir
System	système
Timing diagram	diagramme de timing
Transition	transition
True	vrai
Type	type
Unified Modeling Language (UML)	langage unifié de modélisation (UML)
Unified Process (UP)	Processus Unifié
Use case diagram	diagramme de cas d'utilisation
Use case	cas d'utilisation

Notation graphique

Diagramme d'activités

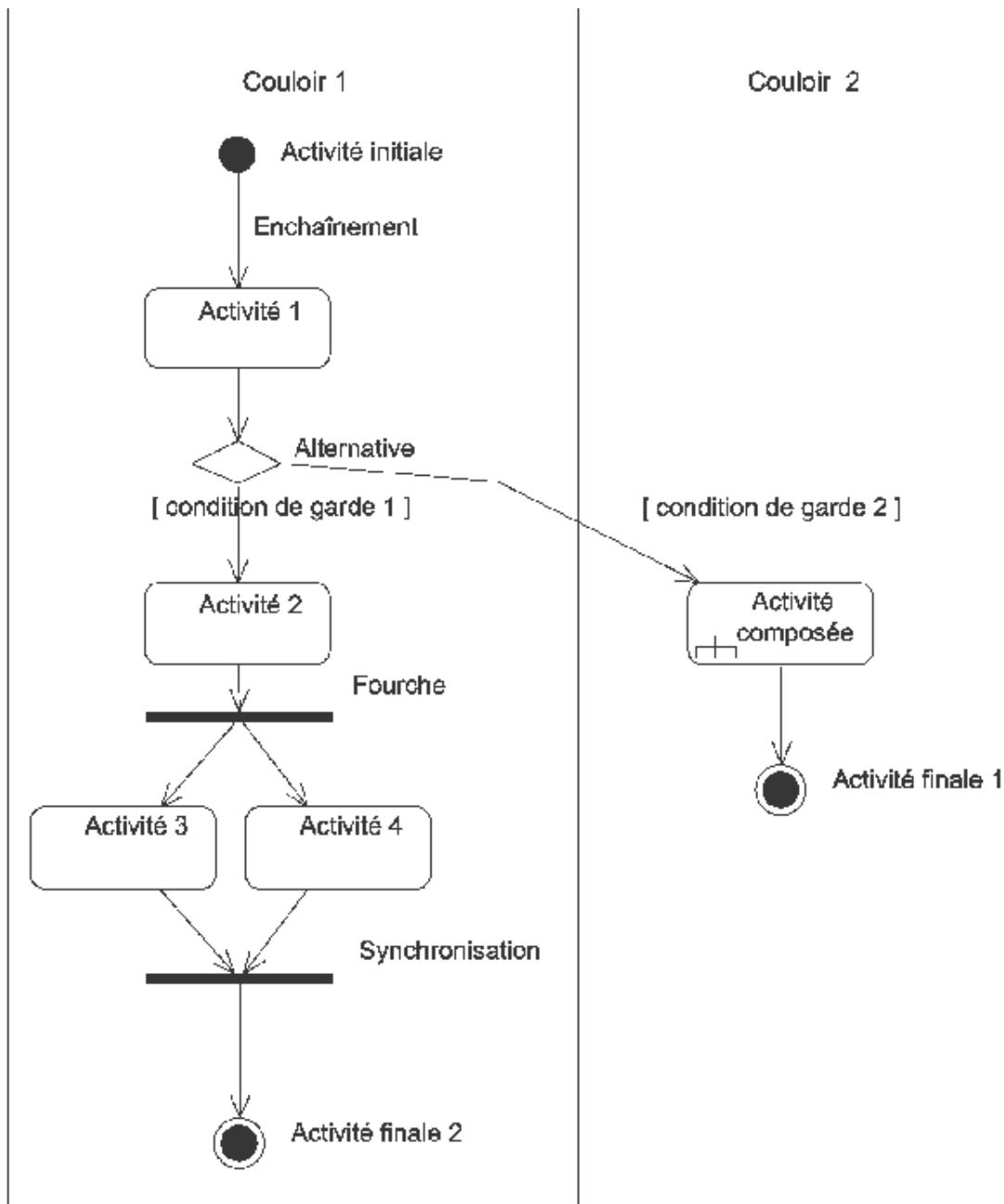


Diagramme de cas d'utilisation

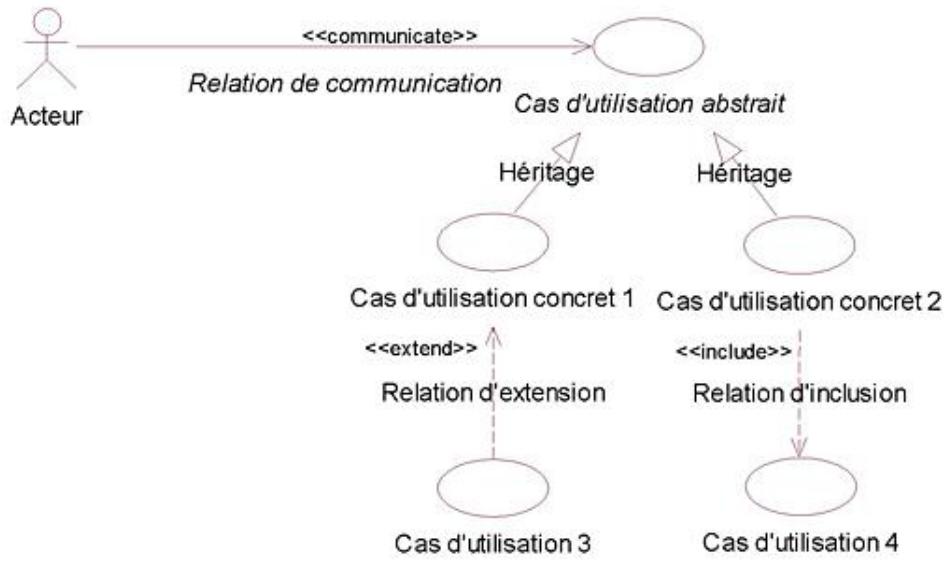


Diagramme de classes

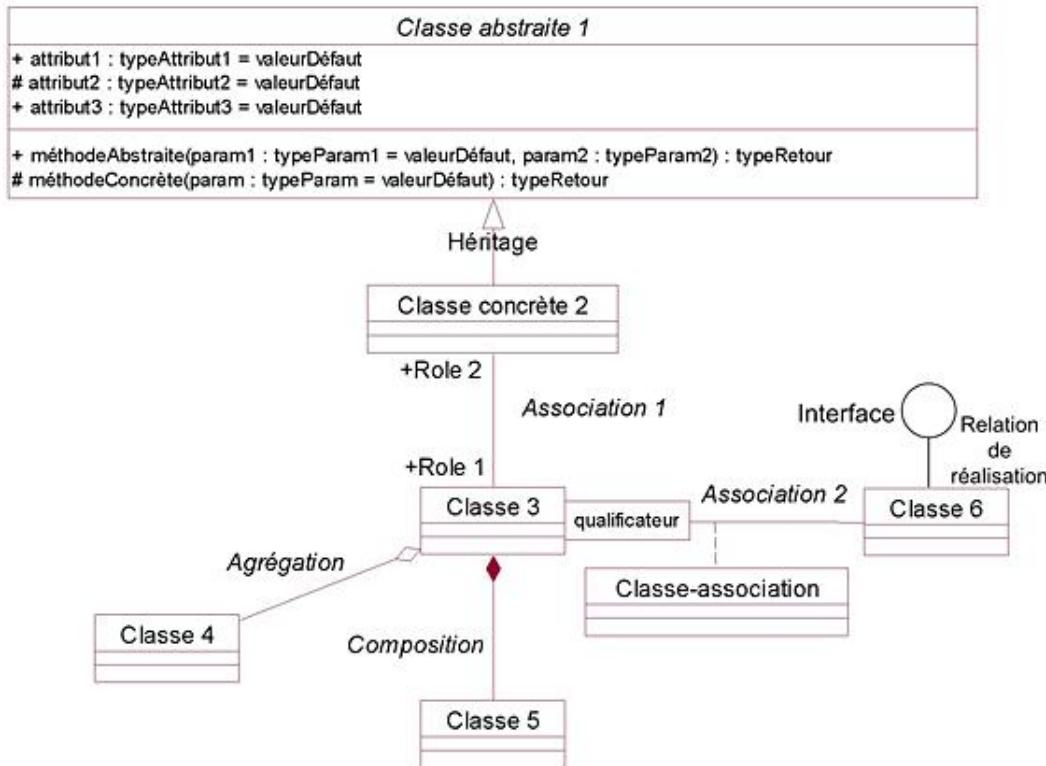
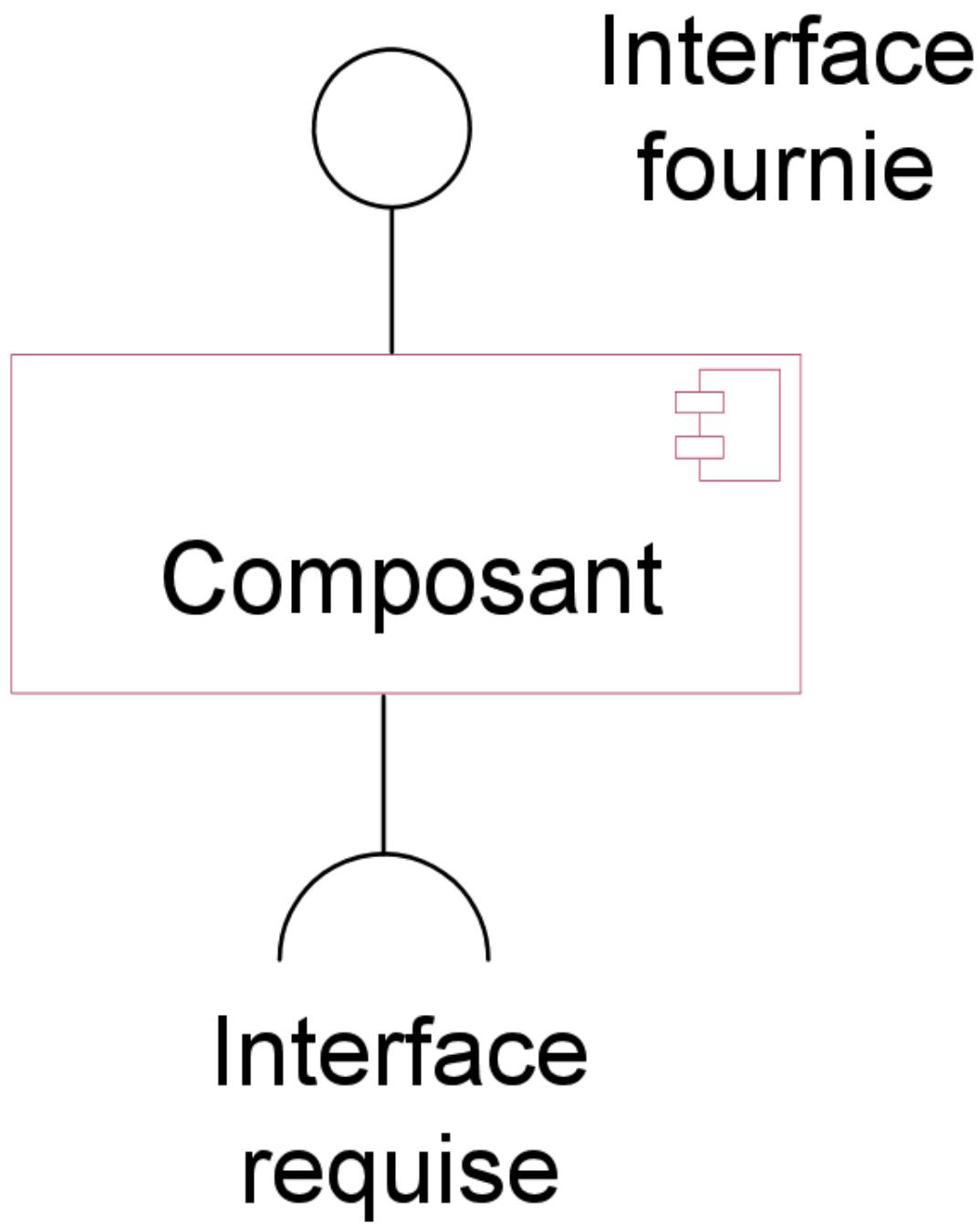
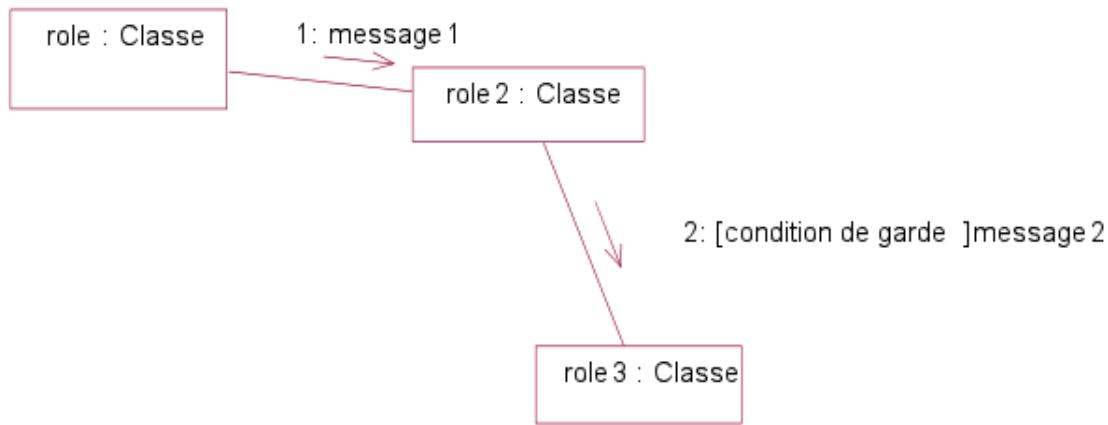


Diagramme de communication



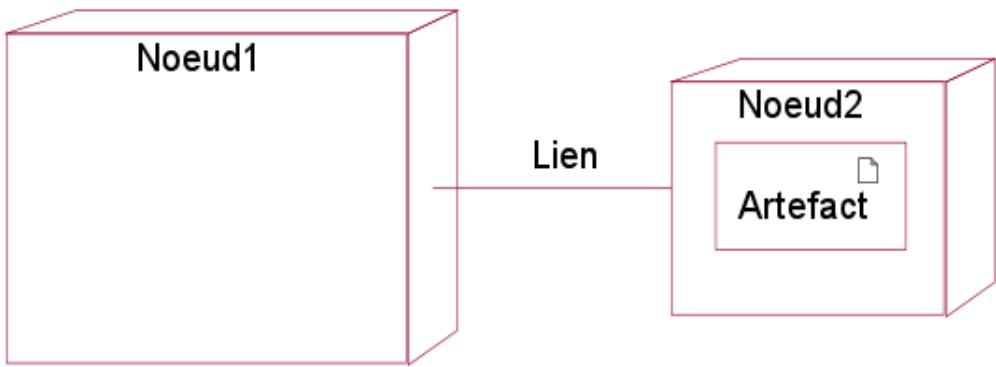


Diagramme d'états-transitions

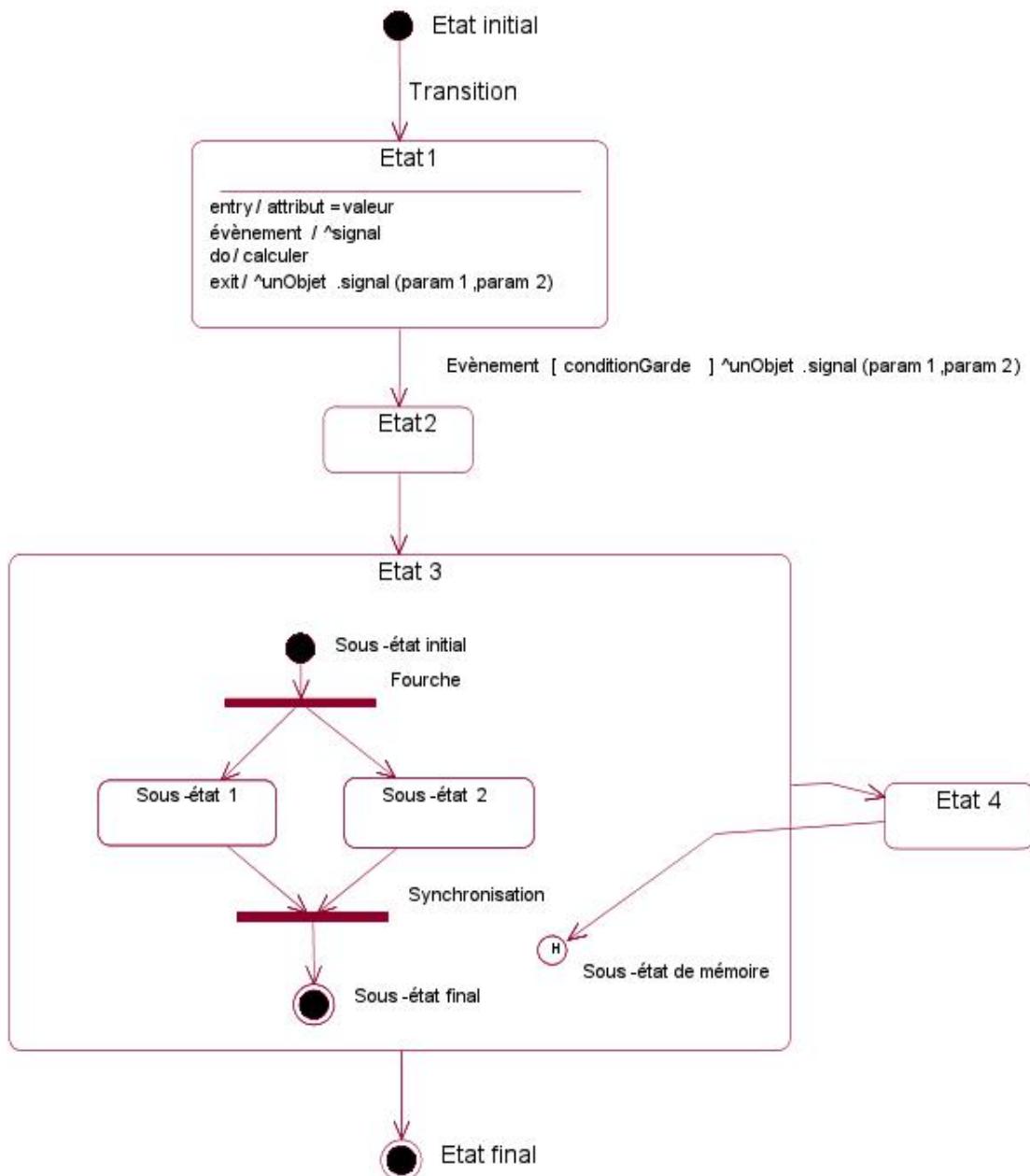
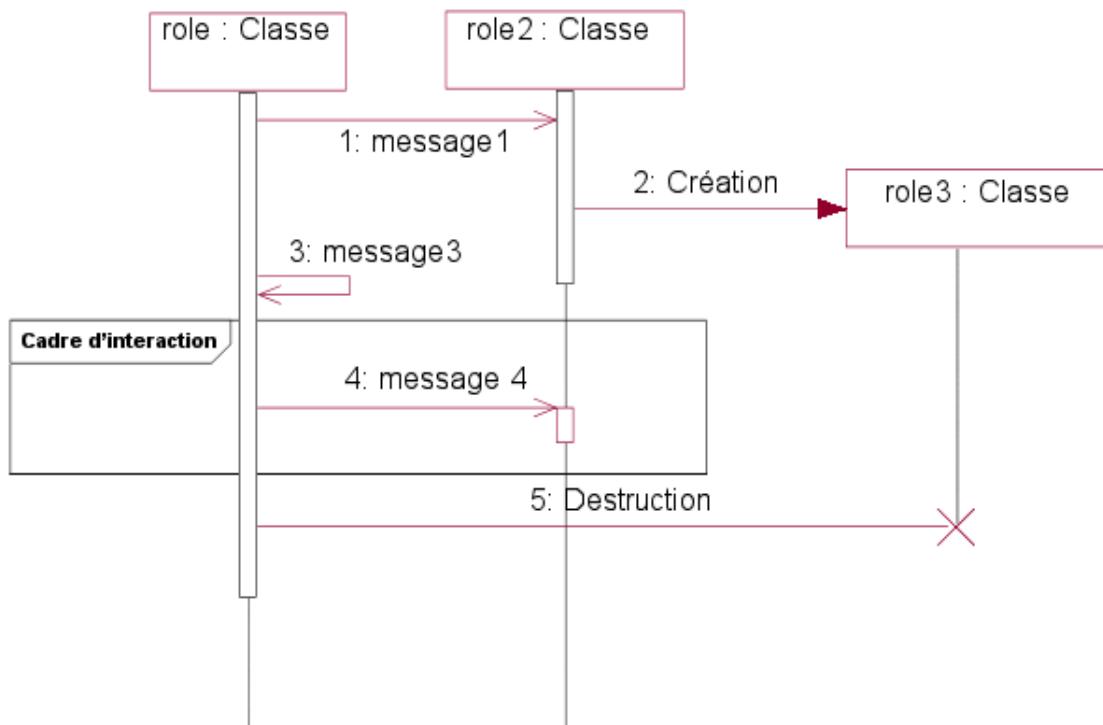


Diagramme de séquence



Bibliographie

- Alistair Cockburn, *Rédiger des cas d'utilisation efficaces*, Eyrolles, 1999
- Anneke Kleppe, Jos Warmer, Wim Bast, *MDA Explained: The Model Driven Architecture--Practice and Promise*, Addison-Wesley, 2003
- Ivar Jacobson, Grady Booch, James Rumbaugh, *Le Processus unifié de développement logiciel*, Eyrolles, 2000
- James Rumbaugh, Ivar Jacobson, Grady Booch, *Unified Modeling Language Reference Manual*, Second Edition, Addison-Wesley, 2004
- Jos Warmer, Anneke Kleppe, *The Object Constraint Language: Getting Your Models ready for MDA*, Second Edition, Addison-Wesley, 2003
- Kendal Scott, *Fast Track UML 2.0*, Apress, 2004
- Object Management Group, *UML 2.0 Infrastructure Final Adopted Specification*, ptc/03-09-15
- Object Management Group, *UML 2.0 Superstructure Final Adopted Specification*, ptc/03-08-02
- Philippe Kruchten, Per Kroll, *Guide pratique du RUP*, CampusPress, 2003
- Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise, *MDA Distilled*, Addison-Wesley, 2004
- Stephen J. Mellor, Marc J. Balcer, Stephen Mellor, Marc Balcer, *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002