Ecole Nationale d'Ingénieurs de Brest

— Cours d'Informatique S4 —

Programmation Orientée Objet (UML)

Cédric BUCHE

buche@enib.fr

version du 25 novembre 2013

Programmation Orientée Objet

— UML —

Cédric Buche

École Nationale d'Ingénieurs de Brest (ENIB)

20 novembre 2013

Ces notes de cours accompagnent les enseignements d'informatique du 4^{ieme} semestre (S4) de Programmation Orientée Objet (POO) de l'Ecole Nationale d'Ingénieurs de Brest (ENIB : `www.enib.fr`). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

Une partie de ce document est librement inspiré de l'ouvrage de Laurent Audibert. Une partie des exercices a été rédigée par Pierre Chevaillier.

Table des matières

Cours S4

1 Introduction

Sommaire

1.1 Objectifs du cours – prérequis

Les objectifs du cours en S4 sont :
▷ Connaître le langage de modélisation UML
▷ Comprendre la sémantique des principaux éléments des différents modèles

Le prérequis est la maîtrise des principes de la programmation orientée objet.

1.2 Importance de la modélisation

Prenons l'image de la niche, la maison familiale et l'immeuble Pour construire une niche, il suffit de quelques planches, des clous, un marteau et quelques outils. Pour une une maison familiale, l'entrepreneur a besoin de plans généraux et de plans d'exécution détaillés (pièces, électricité, plomberie, chauffage). Enfin, pour construire une immeuble, il faudra une planification détaillée, de nombreux plans et études.

Pour une programme informatique, c'est la même chose !

1.3 Pourquoi modéliser ?

La modèlisation permet de mieux comprendre le système en développement. Les objectifs sont importants :

Définition 1.1. *UML : (en anglais Unified Modeling Language, " langage de modélisation unifié ") est un langage graphique de modélisation des données et des traitements. C'est une formalisation très aboutie et non-propriétaire de la modélisation objet utilisée en génie logiciel.*

- ▷ nous aider à le visualiser tel qu'il est ou tel qu'il devrait être
- ▷ spécifier la structure et le comportement d'un système
- ▷ avoir un "patron" pour guider la construction du système
- ▷ documenter les décisions qui ont été prises

Nous construisons des modèles de systèmes complexes parce que nous sommes incapables d'appréhender ces systèmes dans leur entièreté.

1.4 Modélisation informatique : des logiciels au génie logiciel

Les logiciels peuvent être développés par une personne seule, une petite équipe, ou un ensemble d'équipes coordonnées. Le développement de grands logiciels par de grandes équipes pose d'importants problèmes de conception et de coordination. En 1995, une étude dressait un tableau accablant de la conduite des projets informatiques :

- ▷ 16 % des projets étaient conformes aux prévisions initiales,
- ▷ 53 % avaient subi des dépassements en coût et délai d'un facteur 2 à 3 avec diminution du nombre des fonctions offertes,
- ▷ 31,1% ont été abandonnés durant leur développement.

L'examen des causes de succès et d'échec est instructif : la plupart des échecs proviennent non de l'informatique, mais de la maîtrise d'ouvrage (i.e. le client). Pour ces raisons, le développement de logiciels dans un contexte professionnel suit souvent des règles strictes encadrant la conception et permettant le travail en groupe et la maintenance du code. Ainsi, une nouvelle discipline est née : le génie logiciel. Pour apporter une réponse à tous ces problèmes, le génie logiciel s'intéresse particulièrement à la manière dont le code source d'un logiciel est spécifié puis produit. Ainsi, le génie logiciel touche au cycle de vie des logiciels :

- ▷ l'analyse du besoin,
- ▷ l'élaboration des spécifications,
- ▷ la conceptualisation,
- ▷ le développement,
- ▷ la phase de test,
- ▷ la maintenance.

1.5 Introduction à UML

UML est un langage graphique de modélisation pour spécifier, concevoir, construire et documenter des applications informatiques. UML est considéré comme une synthèse des bonnes pratiques de l'ingénierie informatique. Il permet d'unifier des modèles en se basant sur une standardisation (par l'OMG).

UML a pour objectif de fournir un langage visuel et expressif, des mécanismes d'extension, d'être indépendant des technologies et langages d'implémentation. Il fournit une base formelle pour la modélisation.

Dans une approche classique, un projet suivra les étapes suivantes :

1. Fonctionnel
   - ▷ Définition du cahier des charges : le diagramme de *cas d'utilisation* permet alors de produire des scénarios écrits
   - ▷ Élaboration des scénarios formels : les diagrammes de *sequence / communication* permettent d'identifier les objets/classes
2. Statique
   - ▷ Définir le diagramme des *classes*
3. Dynamique
   - ▷ Dynamique de chaque objet est représentée par le diagramme d'*états/transitions*
   - ▷ Dynamique globale du système est représentée par le diagramme le diagramme d'*activités*

Remarque 1.1. *UML 2.0 comporte 13 types de diagrammes.*

UML peut être utilisé pour :
- ▷ Conception (≪ forward engineering ≫)
- ▷ Rétro conception (≪ reverse engineering ≫)
- ▷ Documentation d'un système

Définition 1.2. *OMG : L'Object Management Group est une association américaine à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à la base des standards UML.*

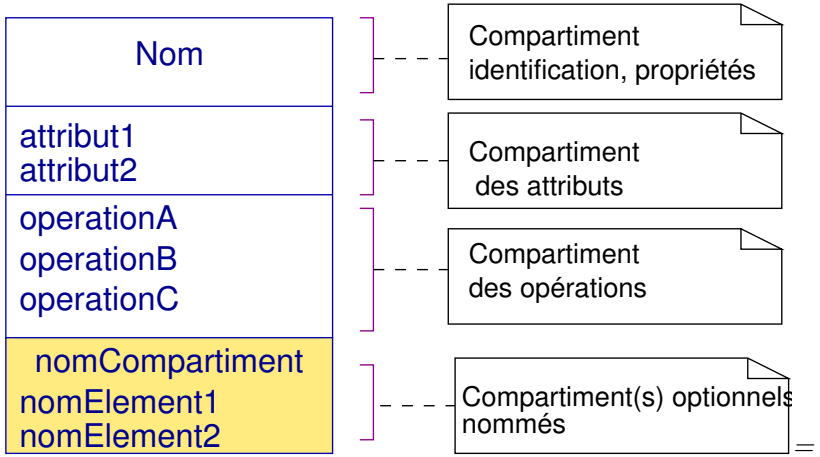
2 Diagramme de classes (4 UC)

Sommaire

2.1 Classes

Une classe se représente par un rectangle constitué de 4 compartiments.

Figure 2.1. *Classe, attribut et opération : notations*

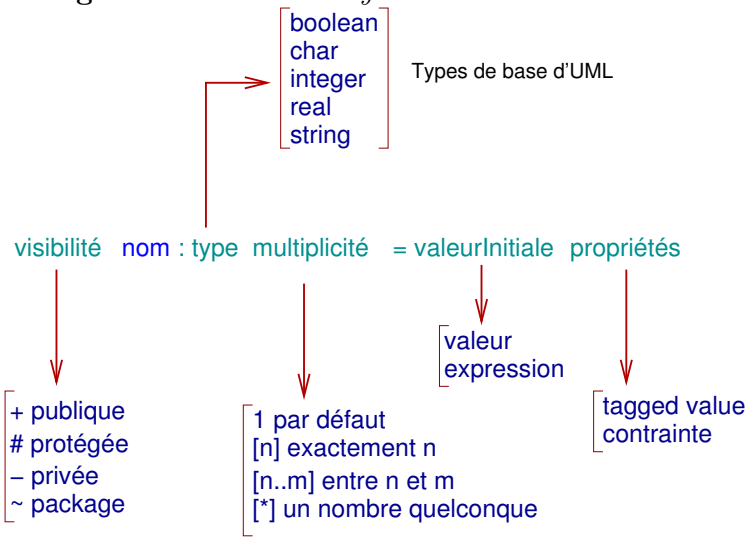


Le nom de la classe commence par une majuscule. Le dernier compartiment est très peu utilisé en pratique.

2.1.1 Attributs

Les attributs définissent des informations qu'une classe ou un objet doivent connaître. Ils représentent les données encapsulées dans les objets de cette classe. Chacune de ces informations est définie par un nom, un type de données, une visibilité et peut être initialisé. Le nom de l'attribut doit être unique dans la classe. La syntaxe de la déclaration d'un attribut est la suivante :

Figure 2.2. *Attribut : syntaxe*



La notion de visibilité est détaillée ici :

+	`public`	tout élément qui accède à la classe
#	`protected`	seul un élément de la classe ou de ses descendants
-	`private`	seul un élément de la classe
~	`package`	seul un élément du même package que la classe

Exemple Voici un exemple :

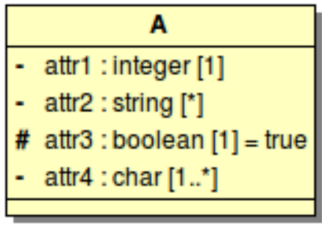


Figure 2.3. *Exemples de classe avec attributs*

Exercices

Exo1 Exprimer sous forme d'un diagramme de classe UML qu'une classe AA possède un attribut attr1, dont l'accés est protégé et qui a comme valeur initiale 1.0

Exo2 Exprimer sous forme d'un diagramme de classe UML qu'une classe AB possède un attribut xyz, librement accessible qui est une série de chaînes de caractères

Mediathèque Dans une mediathèque, une œuvre est referencée par un code d'identification. Il est possible de faire une recherche par le nom des auteurs, ou par le titre de l'œuvre. Écrire le digramme de classe correspondant.

2.1.2 Opérations

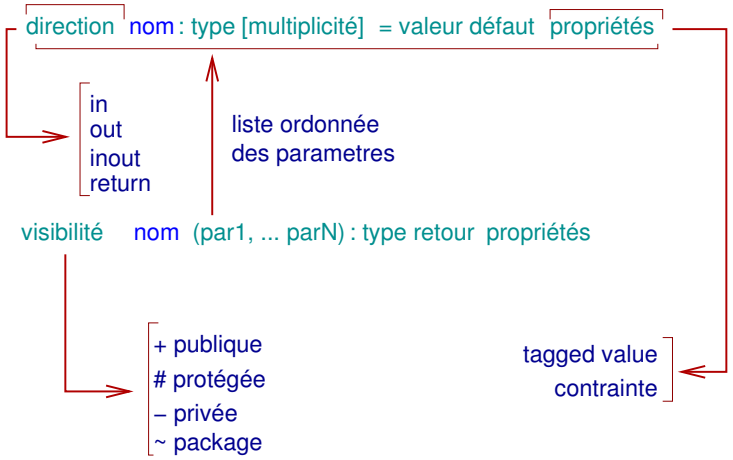


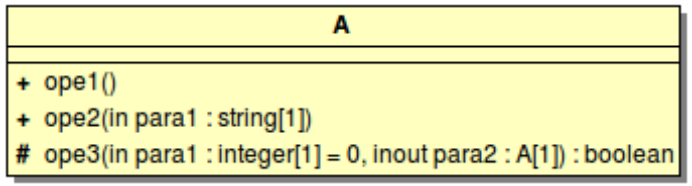
Figure 2.4. *Opération : syntaxe*

Direction des paramètres Plusieurs directions sont possibles :

- ▷ information que l'objet serveur ne possède pas, mais qui est nécessaire à la réalisation de l'opération : direction = in
- ▷ information nécessaire à la réalisation de l'opération et transformée par celle-ci : direction = inout
- ▷ information produite par l'exécution de l'opération, donc inexistante avant ; direction = out ou return

Exemple Voici un exemple de classe avec opérations :

Figure 2.5. *Exemple d'opérations UML*



Exercices

Exo1 Exprimer sous forme d'un diagramme de classe UML que les objets d'une classe AX sont composés d'objects de la classe AY ; ces objets, nommés abc sont d'accés protégé et sont en nombre quelconque. On ne représentera pas la classe AY sur le diagramme.

Exo2 Exprimez sous forme d'un diagramme de classe UML qu'une classe AA possède les propriétés suivantes :

- ▷ un attribut attr1, de type réel, dont l'accès est protégé et qui a comme valeur initiale 1.0.
- ▷ un attribut xyz, librement accessible qui est une série de chaînes de caractères.
- ▷ une opération ope de visibilité protégée qui utilise un entier val (non modifié), un objet obj de la classe AB qu'elle modifie, et qui retourne une valeur booléenne.

On ne représentera pas la classe AB sur le diagramme.

Exo3 Exprimez sous forme d'un diagramme de classe UML qu'une classe CA possède les propriétés suivantes :

- ▷ Un attribut a1, de type entier, dont l'accès est protégé et dont la valeur initiale est 0.
- ▷ un attribut x, de type réel, dont l'accès est privé, qui est non modifiable, et qui a comme valeur initiale 1.0.
- ▷ une opération foo de visibilité protégée qui utilise un entier val (non modifié), et qui retourne une valeur booléenne.
- ▷ une opération moo de visibilité privée qui utilise un ensemble stuff d'objets de la classe AB qu'elle modifie.

On ne représentera pas la classe AB sur le diagramme.

QCM

1. Dans la représentation graphique d'une classe, les attributs... _____

   (a) figurent dans le compartiment juste en dessous de celui du nom de la classe
   (b) figurent dans le compartiment du bas
   (c) figurent aux extrémités des relations
   (d) sont écrits en bleu
   (e) ne sont pas représentables

2. La syntaxe correcte d'un attribut `attr` est : _____

   (a) public int attr
   (b) + attr: integer
   (c) public attr: int
   (d) int _attr
   (e) +: attr integer
   (f) + integer attr

3. Quelle liste ne contient que des types de base UML ? _____

   (a) bool, char*, int, double
   (b) boolean, string, integer, real
   (c) bool, char, int, float
   (d) bool, string, int, float

4. En UML, les arguments d'une méthode s'appellent : _____

   (a) les attributs d'une opération
   (b) les arguments d'une opération
   (c) les paramètres d'une opération
   (d) les paramètres d'une méthode
   (e) les attributs d'une méthode
   (f) les arguments d'une méthode

2.1.3 Constructeurs et destructeurs

Comment représenter un constructeur et un destructeur dans un diagramme de classes UML ?
La solution est d'utiliser les stéréotypes « create » et « destroy » pour les représenter

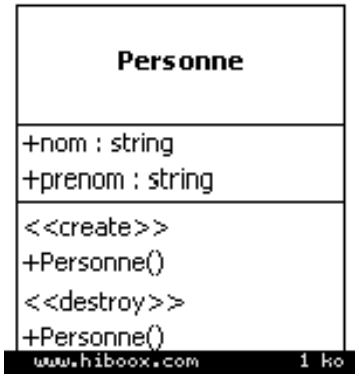


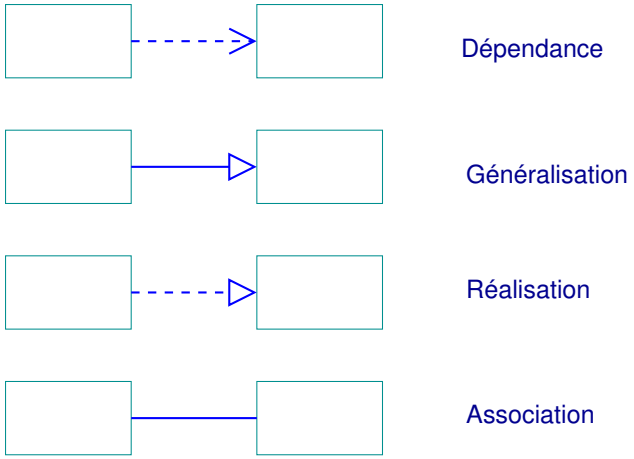
Figure 2.6. *Exemple d'opérations UML*

2.2 Relations entre classes

2.2.1 Types

Il existe 4 types de relations liant plusieurs classes :

Figure 2.7. *Différentes relations entre classes*



2.2.2 Dépendance

Présentation

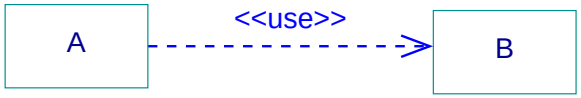


Figure 2.8.

La relation de dépendance indique une dépendance entre les propriétés d'une classe (le *client*) et une autre classe (le *serveur*, *supplier*). En conséquence, une modification du *serveur* peut affecter le comportement du *client*.

La relation de dépendance est représentée par un trait discontinu orienté. Elle indique que la modification de la cible peut impliquer une modification de la source.

Exemples

- une opération de la classe `A` fait appel à une opération de la classe `B`
- une opération de `A` a comme paramètre un objet `B`

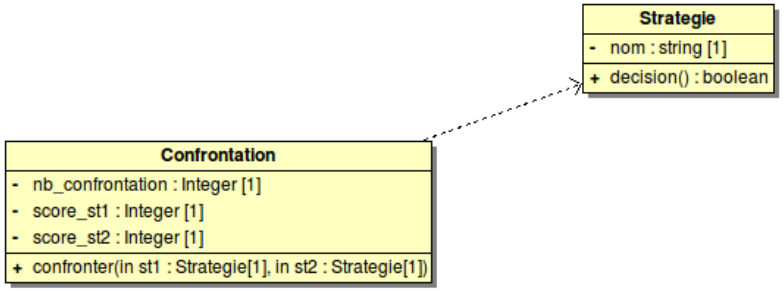


Figure 2.9.

On utilise souvent une dépendance quand une classe en utilise une autre comme argument dans la signature d'une opération. Par exemple, le diagramme de la figure montre que la classe `Confrontation` utilise la classe `Stratégie` car la classe `Confrontation` possède une méthode confronter dont deux paramètre sont du type `Stratégie`. Si la classe `Stratégie`, notamment son interface, change, alors des modifications devront également être apportées à la classe `Confrontation`.

Stéréotypes La dépendance est souvent stéréotypée pour mieux expliciter le lien sémantique entre les éléments du modèle.

`access`	import du contenu d'un autre package
`create`	la classe crée des instances d'une autre classe
`instantiate`	la méthode d'une classe crée des instances d'une autre
`permit`	donne accès aux éléments privés
`use`	un élément requiert un autre élément

2.2.3 Héritage

Présentation

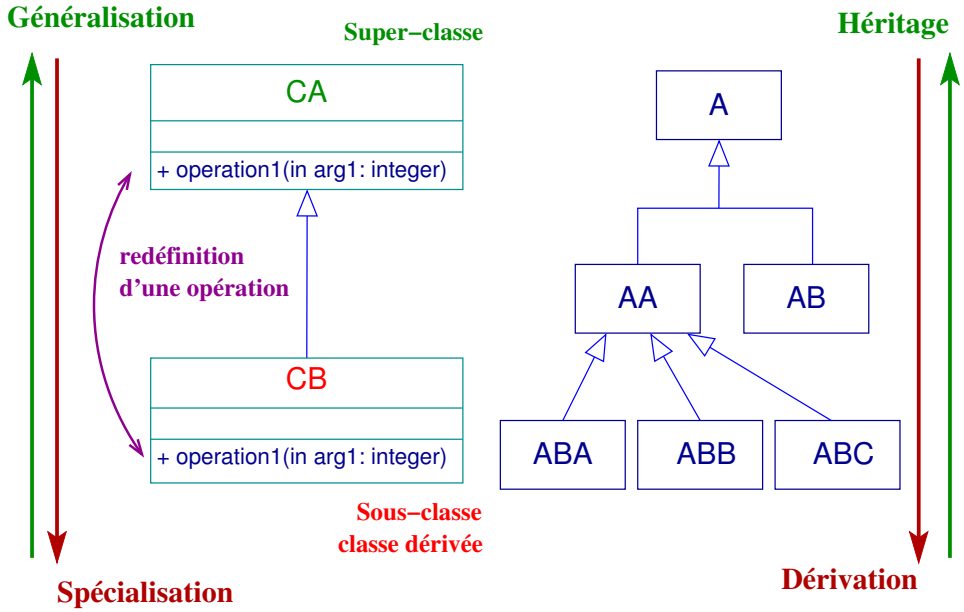


Figure 2.10.

Dans une relation de généralisation entre classes, la super-classe est la classe *parent* et la sous-classe, la classe *enfant*. Partout où une instance du parent est utilisée, une instance de l'enfant est aussi utilisable (principe de substitution).

Une instance de la classe CB a comme type direct CB et comme type indirect CA.

La classe spécialisée est intégralement cohérente avec la classe de base, mais comporte des informations supplémentaires (attributs, opérations, associations). Un objet de la classe spécialisée peut être utilisé partout où un objet de la classe de base est autorisé.

Le symbole utilisé pour la relation d'héritage ou de généralisation est une flèche avec un trait plein dont la pointe est un triangle fermé désignant le cas le plus général.

Exemple Voici un exmple d'héritage entre la classe `Oeuvre` et les classes `Opera, Livre, Film`

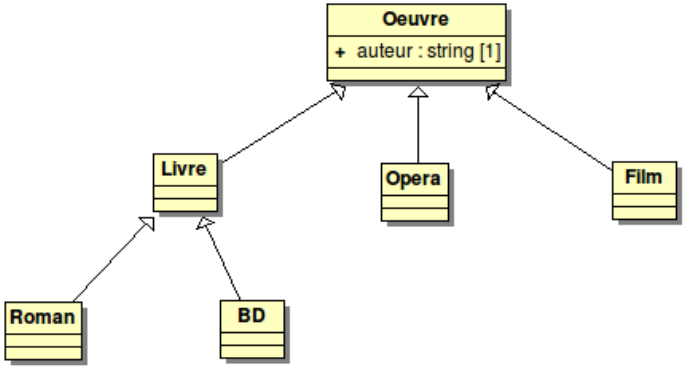


Figure 2.11.

Vocabulaire Plusieurs expressions sont utilisées :

Figure 2.12.

A	est une spécialisation	de B
A	est une sous-classe	de B
A	dérive	de B
A	hérite	de B
B	est une généralisation	de A
B	est une super-classe	de A

Exercices

Concepts On considère les classes `CA` et `CB` ayant les opérations suivantes.

1. La classe `CA` a deux opérations privées `meth1()` et `meth2()`.
2. La classe `CA` a une opération protégée `init()`.
3. La classe `CA` a une opération publique `doIt()`.
4. La classe `CB` est une spécialisation de `CA`.
5. L'opération `meth2` est redéfinie dans la classe `CB`.
6. La classe `CB` a une opération public `foo()`.

Représenter ce modèle par un diagramme de classes UML.

L'opération `doIt` fait appel aux opérations `init()`, `meth1()` et `meth2()`.

Est-ce correct et quelles méthodes sont exécutées si l'on fait appel à cette opération sur un objet de la classe CA ? Même question dans le cas d'un objet de la classe `CB`.

L'opération `foo` fait également appel aux opérations `init()`, `meth1()` et `meth2()`.

Est-ce correct et quelles méthodes sont exécutées si l'on fait appel à cette opération sur un objet de la classe CB ?

Accés On considère les éléments suivants d'un modèle de classe :

1. Une classe CX qui a comme opération
   - ▷ alpha de visibilité privée
   - ▷ beta de visibilité protégée
   - ▷ gamma de visibilité public
2. Une classe concrète CY
   - ▷ ayant une opération protégée omega

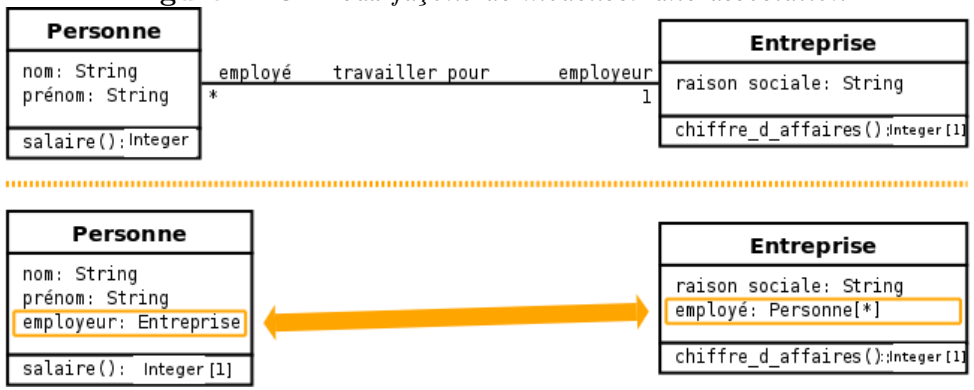
▷ et dont CX est une généralisation

Représentez ces éléments sous forme d'un diagramme de classe UML.
Quelle(s) opération(s) peut-on appeler sur un objet de la classe CY ?
Quelle(s) opération(s) de CX un objet de la classe CY peut-il appeler dans le cadre de son opération omega ?

2.2.4 Association

Une association est une relation entre deux classes (association binaire) ou plus (association n-aire), qui décrit les connexions structurelles entre leurs instances. Une association indique donc qu'il peut y avoir des liens entre des instances des classes associées.

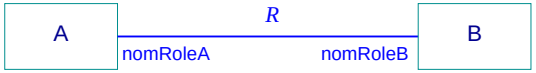
Figure 2.13. *Deux façons de modéliser une association*



Un attribut n'est donc rien d'autre qu'une terminaison d'un cas particulier d'association

Rôle Comme un attribut, une terminaison d'association peut être nommée. Le nom est situé à proximité de la terminaison, mais contrairement à un attribut, ce nom est facultatif. Le nom d'une terminaison d'association est appelée nom du rôle. Une association peut donc posséder autant de noms de rôle que de terminaisons (deux pour une association binaire et n pour une association n-aire).

Figure 2.14. *Notion de rôle*



nomRole : indique ce que représente l'ensemble des instances associées à une instance de la classe par la relation `R`.

nomRoleB nom de l'ensemble des instances de la classe `B` qui sont en relation avec `1` instance de la classe `A` par la relation R.

nomRoleA nom de l'ensemble des instances de la classe `A` qui sont en relation avec `1` instance de la classe `B` par la relation R.

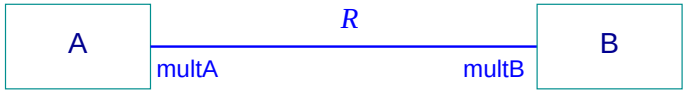
Remarque : l'information concernant le rôle est portée par l'extrémité de la relation.

Visibilité Comme un attribut, une terminaison d'association possède une visibilité. La visibilité est mentionnée à proximité de la terminaison, et plus précisément, le cas échéant, devant le nom de la terminaison.

L'accès peut être `+`, `#` ou `-`

Multiplicité Comme un attribut, une terminaison d'association peut posséder une multiplicité. Elle est mentionnée à proximité de la terminaison. Il n'est pas impératif de la préciser, mais, contrairement à un attribut dont la multiplicité par défaut est 1, la multiplicité par défaut d'une terminaison d'association est non spécifiée. L'interprétation de la multiplicité pour une terminaison d'association est moins évidente que pour un attribut.

Figure 2.15. *Valeurs possibles du cardinal de l'ensemble des instances associées à une instance de la classe par la relation* `R`.



multB cardinal de l'ensemble des instances de la classe `B` qui sont en relation avec `1` instance de la classe `A` par la relation R.

multA cardinal de l'ensemble des instances de la classe `A` qui sont en relation avec `1` instance de la classe `B` par la relation R.

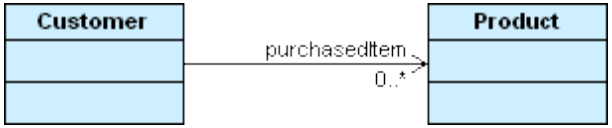
Multiplicité : notation

notation : `min .. max`

	abreg.	signification
1..1	**1**	exactement 1
0..1	–	zéro ou un (optionnel)
0..*	*****	aucun ou plusieurs
1..*	–	au moins 1
n..m, p	–	entre n et m ou exactement p

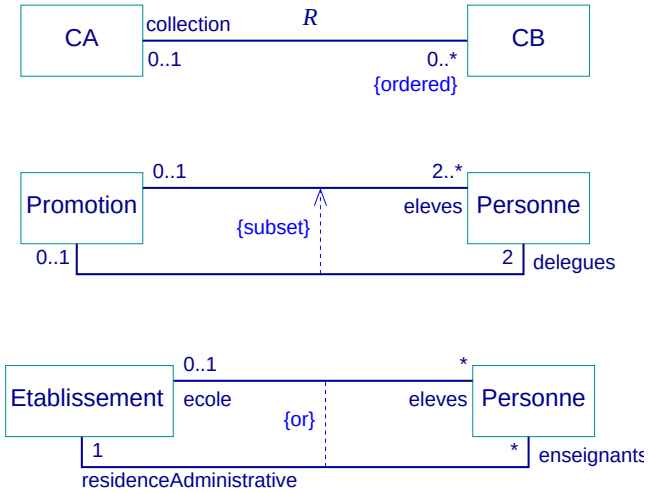
Association unidirectionnelle L'association peut être unidirectionnelle :

Figure 2.16. *Exemples d'association unidirectionnelle*



Contraintes sur une association Les associations peuvent être contraintes :

Figure 2.17. *Exemples de contraintes*



Exercices

Exo1 Exprimez sous forme d'un diagramme de classe UML qu'une classe Line est composée de deux Point. Exprimer également que la classe Line propose un service pour se déplacer d'un dx et dy, et un service qui retourne la distance à une Line.

Syntaxe Les objets de la classe AA référencent un ensemble setOfStuff d'objets de la classe AB qui sont d'accès public. Chaque objet de AB référence, par la même relation, un objet, ou aucun, de la classe AA, nommé xyz, d'accès protégé.

Représentez ces éléments sous la forme d'un diagramme de classe UML.

Répertoires Dans un système de fichiers, géré par le système d'exploitation d'un ordinateur, un répertoire peut contenir des répertoires.

Représentez cette structure sous la forme d'un diagramme de classe UML.

Dessin On s'intéresse à un logiciel de dessin technique et plus particulièrement au tracé de polygones. Un polygone est défini par la liste de ses sommets (au moins deux). S'agissant de dessin en 2D, les sommets sont des points caractérisés par leurs coordonnés : (x, y).

Le tracé d'un polygone est fonction de l'ordre dans lequel ses sommets sont référencés ; cet ordre doit donc être connu. Pour certaines opérations, on a aussi besoin de référencer le point correspondant au barycentre d'un polygone.

Les arêtes du polygone sont des segments qui relient 2 points, nommés respectivement origine et cible de l'arête.

Exprimez sous forme d'un diagramme de classe UML ces différentes propriétés relatives aux polygones.

2.2.5 Agrégation et composition

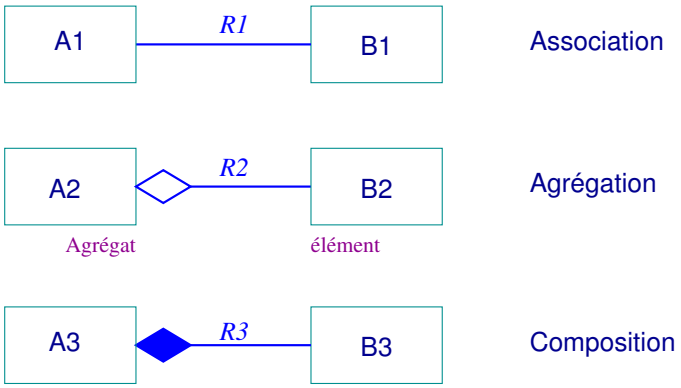


Figure 2.18.

Agrégation Une association simple entre deux classes représente une relation structurelle entre pairs, c'est à dire entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre. Lorsque l'on souhaite modéliser une relation tout/partie où une classe constitue un élément plus grand (tout) composé d'éléments plus petit (partie), il faut utiliser une agrégation.

Une agrégation est une association qui représente une relation d'inclusion structurelle ou comportementale d'un élément dans un ensemble. Graphiquement, on ajoute un losange vide du côté de l'agrégat. Contrairement à une association simple, l'agrégation est transitive.

La signification de cette forme simple d'agrégation est uniquement conceptuelle. Elle ne contraint pas la navigabilité ou les multiplicités de l'association. Elle n'entraîne pas non plus de contrainte sur la durée de vie des parties par rapport au tout.

Composition La composition, également appelée agrégation composite, décrit une contenance structurelle entre instances. Ainsi, la destruction de l'objet composite implique la destruction de ses composants. Une instance de la partie appartient toujours à au plus une instance de l'élément composite : la multiplicité du côté composite ne doit pas être supérieure à 1 (i.e. 1 ou 0..1). Graphiquement, on ajoute un losange plein du côté de l'agrégat.

Exemples Voici un exemple :

Figure 2.19. *Exemple de relation d'agrégation et de composition.*

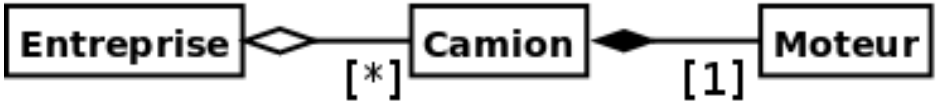
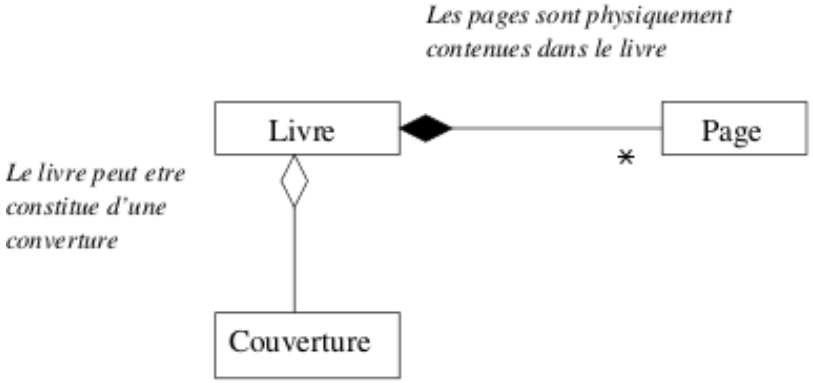


Figure 2.20. *Exemple de relation d'agrégation et de composition.*

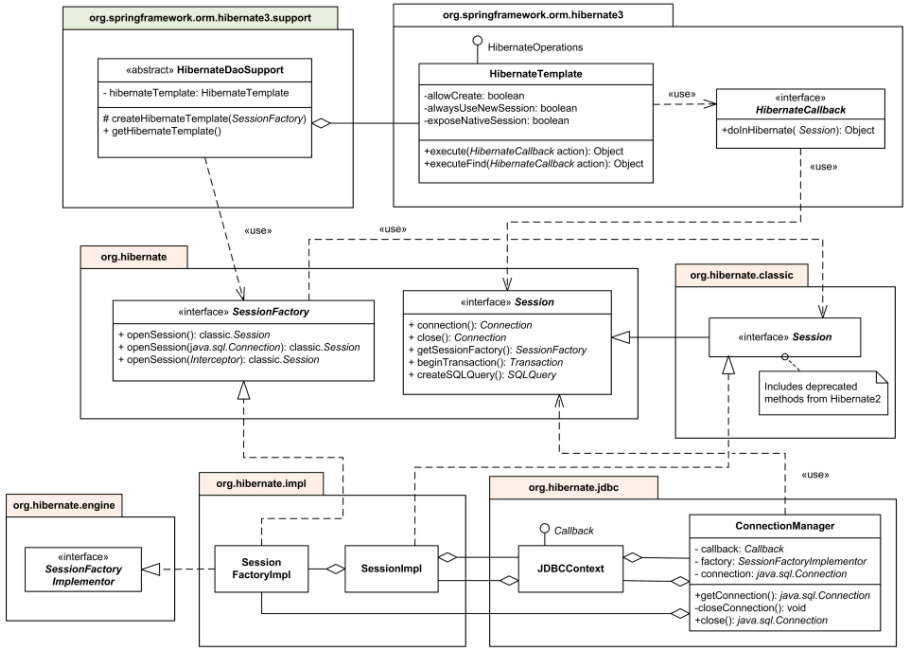


2.2.6 Package

Principe Un package en UML (ou paquetage en français) est un groupe d'éléments, dans le but de les grouper dans des ensembles cohérents. Un package peut contenir la plupart des éléments UML : classes, objets, cas d'utilisations, composantes, etc. Il peut également contenir des packages, créant une hiérarchie complète. L'avantage des packages est qu'ils permettent de structurer les diagrammes et donnent une vision globale plus claire.

Exemple

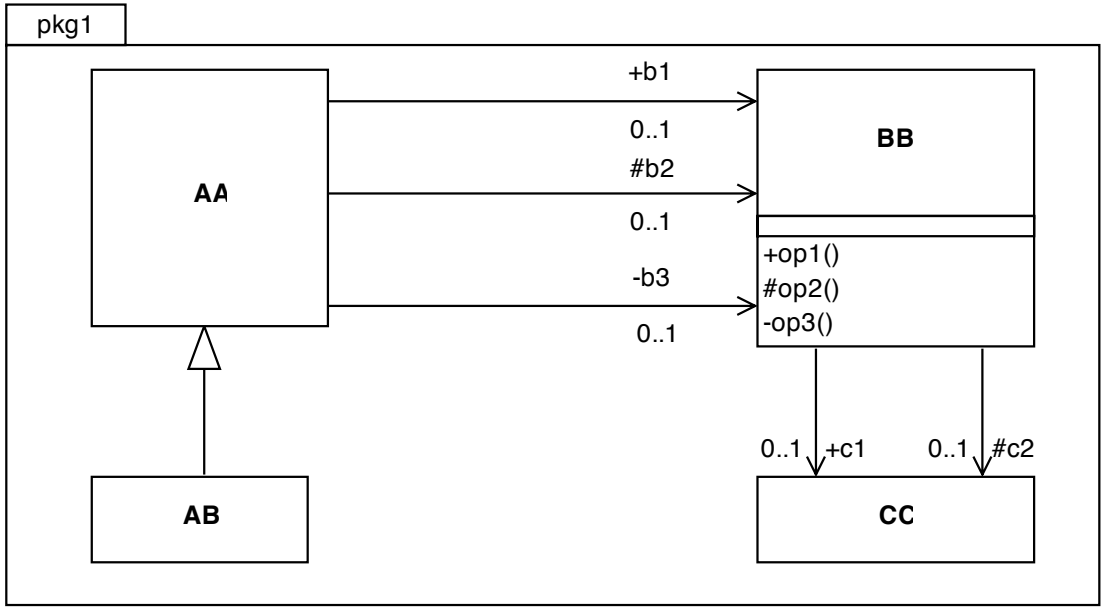
Figure 2.21. *Une diagramme structuré par plusieurs packages.*



Exercices

Notation Exprimez sous forme d'un diagramme de classe UML que le package PP contient :
- ▷ une classe A1
- ▷ une classe A2

2.2.7 Navigation



On considère le modèle de classes du package `pkg1` ci-dessus.

1. Une instance de la classe `AA` peut-elle exécuter les opérations `op1`, `op2` et `op3` de la classe `BB`, repectivement sur `b1`, `b2` et `b3`.
2. Même question pour une instance de `AB`.

3. Quels sont les objets de la classe `CC` qui sont accessibles, et par quelles voies, à une instance de AA ?

4. Même question pour une instance de `AB`.

2.3 Concepts avancés

2.3.1 Classe abstraite

Présentation

- ▷ Définition :
  - ◇ classe non instanciable
  - ◇ ensemble de propriétés communes à différentes classes
    mais partiellement définies.
  - ◇ donc, seuls des objets d'une classe dérivée sont instanciables
- ▷ Deux raisons :
  1. bien que l'instanciation d'un tel objet serait possible cela n'aurait pas de sens
     ex. *Personne* – `Eleve` – `Prof`
  2. au moins une des propriétés de la classe n'est pas définie
     ex. *Shape* opération `draw`
- ▷ Notation :
  - ◇ `<< abstract >>` : tagged-value placée après le nom de la classe
  - ◇ *`NomClass`* : en italique

Exercices

Syntaxe On considère un logiciel permettant de dessiner différents types de graphiques. Il repose sur la notion de « forme graphique », qui correspond à la classe abstraite appelée Shape.

Représentez cette classe abstraite Shape sous forme d'un diagramme de classe UML.

2.3.2 Attribut dérivé

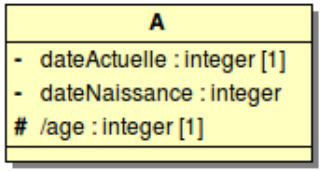
Principe Les attributs dérivés peuvent être calculés à partir d'autres attributs et de formules de calcul. Lors de la conception, un attribut dérivé peut être utilisé comme marqueur jusqu'à ce que vous puissiez déterminer les règles à lui appliquer.

Les attributs dérivés sont symbolisés par l'ajout d'un « / » devant leur nom.

Exemple Exemple de l'attribut `+/surface : real = largeur * longueur`

Exemple de la classe A :

Figure 2.22. *L'attribut* `age` *est calculé à partir de la date de naissance et de la date actuelle*



2.3.3 Interface

Principe

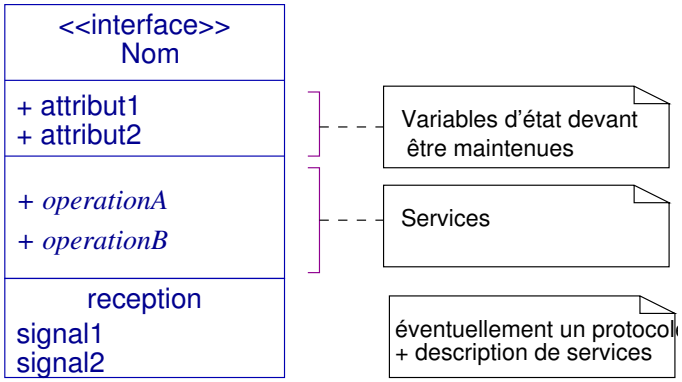


Figure 2.23.

Une interface est une vue externe d'un objet, elle définit les services accessibles (offerts) aux utilisateurs de l'objet.

Le rôle de ce classeur, stéréotypé « interface », est de regrouper un ensemble de propriétés et d'opérations assurant un service cohérent.

Réalisation

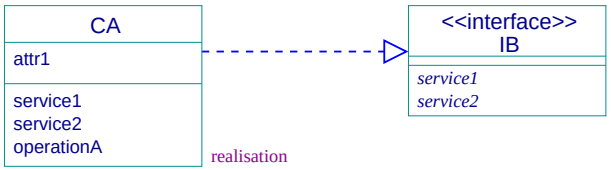


Figure 2.24.

Une interface fournit une vue totale ou partielle d'un ensemble de services offerts par une classe. Les éléments qui utilisent l'interface peuvent exploiter tout ou partie de l'interface.

Dans un modèle UML, le symbole `<<interface>>` sert à identifier de manière explicite et symbolique les services offerts par un élément et l'utilisation qui en est faite par les autres éléments.

Exemple

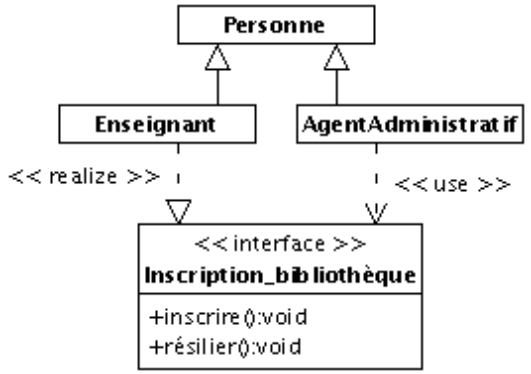


Figure 2.25.

Une interface doit être réalisée par au moins une classe. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire et le stéréotype « realize ». Une classe (classe cliente de l'interface) peut dépendre d'une interface (interface requise). On représente cela par une relation de dépendance et le stéréotype « use ».

Deux types de notation

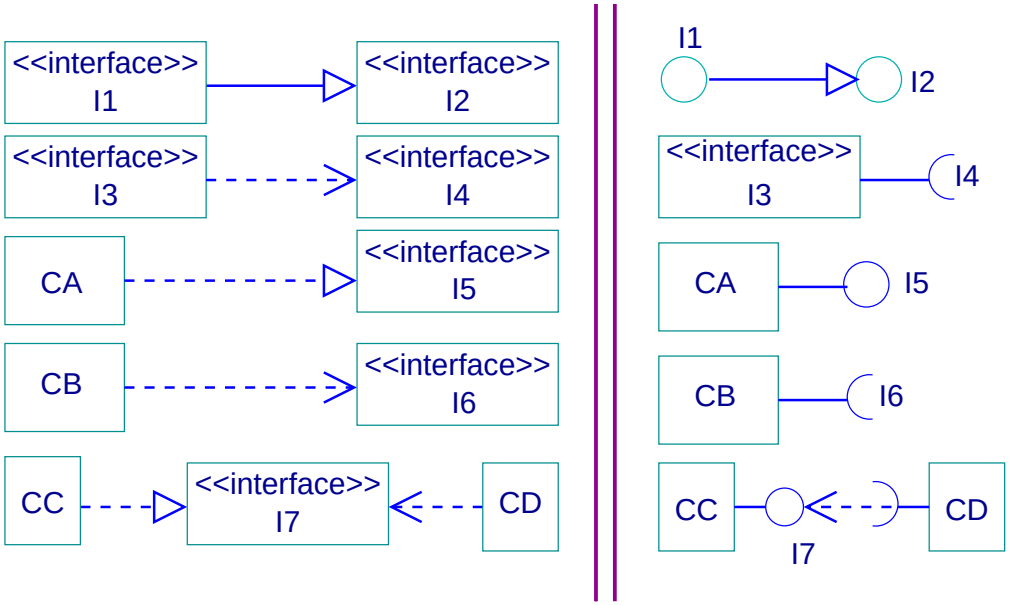
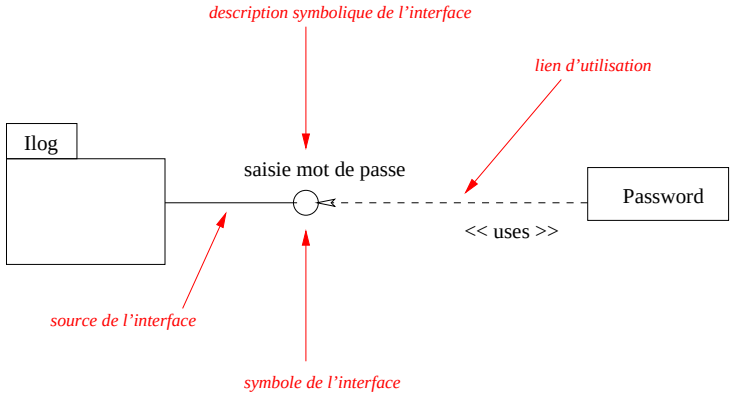


Figure 2.26.

Exemples Voici un exemple :

Figure 2.27.



Exercices

Syntaxe 1 Soit l'interface de nom IXA définissant le service doIt().

Représentez cette interface sous forme d'un diagramme de classe UML.

Syntaxe 2 On considère les éléments suivants d'un modèle de classe :

1. Une interface XY qui définit le service foo ayant comme paramètre d'entrée un entier et retournant un booléen.
2. Une interface ZU qui spécialise XY, et qui définit un service iop ayant comme paramètre d'entrée-sortie un nombre réel et comme paramètre de sortie une chaîne de caractères.
3. Une classe concrète CA qui réalise ZU.

Représentez ces éléments sous forme d'un diagramme de classe UML.

Syntaxe 3 On considère un modèle de classes composé des éléments suivants.

1. Une interface `II` qui définit le service `s1`.
2. Une classe abstraite `AA` qui définit une opération abstraite `ope1` d'accès publique et qui a comme paramètre d'entrée un objet `o` dont la classe réalise l'interface `II`.
3. Une classe `AB` qui spécialise la classe `AA` et qui réalise l'interface `IJ` qui est elle-même requise par la classe `AC`.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

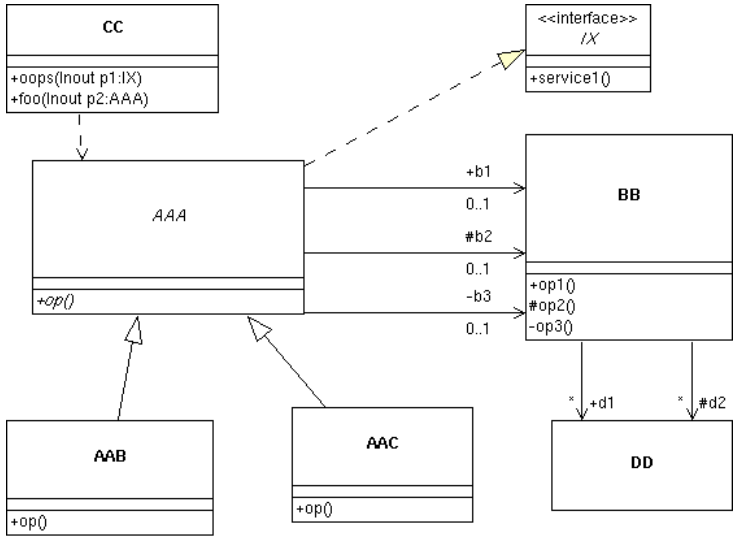
Exo 1 On considère un modèle de classes composé des éléments suivants :

- ▷ une classe abstraite AO qui définit une opération abstraite ope1() (d'acccés publique),
- ▷ une interface I1 qui définit le service s1(),
- ▷ une classe A1 ayant les propriétés suivantes :
  - ◇ elle définit et implémente une opération protégée ope2(),
  - ◇ elle spécialise la classe AO,
  - ◇ elle réalise l'interface I1,

◇ il s'agit d'une classe concrète.

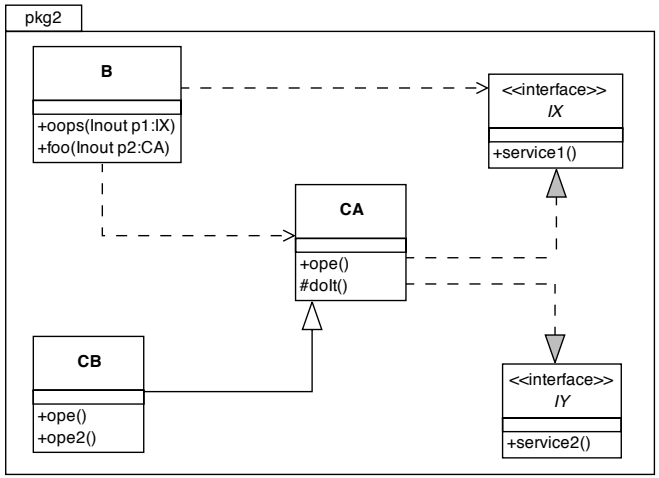
Représentez cette interface sous forme d'un diagramme de classe UML.

Concepts 1 On considère le modèle de classes ci-dessous.



1. Compléter le diagramme afin que l'interface `IX` soit réalisée.
2. L'opération `op` de la classe `AAA` peut-elle exécuter les opérations `op1`, `op2` et `op3` de la classe `BB`, repectivement sur `b1`, `b2` et `b3`.
3. Pour une instance de `AAB`, quels sont les objets de la classe `DD` qui sont accessibles et par quelles voies ?
4. La méthode associée à l'opération `oops` de la classe `CC` a comme paramètre un objet de type `IX`. Quelles opérations de `AAA` peut-elle utiliser ?
5. Même question dans le cas de l'opération `foo`.

Concepts 2

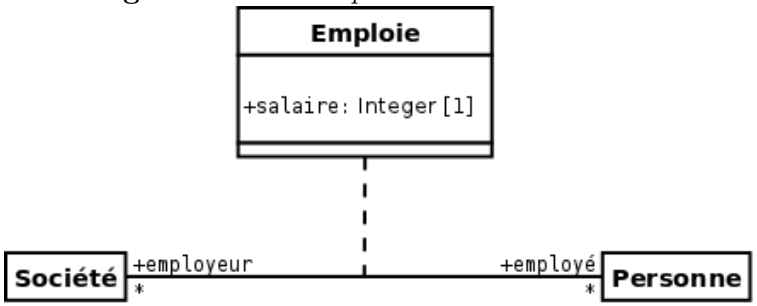


On considère le modèle de classes du package `pkg2` ci-dessus. On s'intéresse à l'utilisation des classes `CA` et `CB`, par une instance de la classe `B` via les opérations `oops` et `foo`.

1. Complétez le diagramme de façon à ce que la classe `CA` implémente effectivement les interfaces `IX` et `IY`.
2. Quelles méthodes des classes `CA` et `CB` peuvent être invoquées par l'opération `foo` sur le paramètre qui lui est fourni ?
3. Même question dans le cas de l'opération `oops`.

2.3.4 Classe-association

Figure 2.28. *Exemple de classe-association.*



Parfois, une association doit posséder des propriétés. Par exemple, l'association `Emploie` entre une société et une personne possède comme propriétés le salaire et la date d'embauche. En effet, ces deux propriétés n'appartiennent ni à la société, qui peut employer plusieurs personnes, ni aux personnes, qui peuvent avoir plusieurs emplois. Il s'agit donc bien de propriétés de l'association Emploie. Les associations ne pouvant posséder de propriété, il faut introduire un nouveau concept pour modéliser cette situation : celui de classe-association.

Une classe-association possède les caractéristiques des associations et des classes : elle se connecte à deux ou plusieurs classes et possède également des attributs et des opérations.

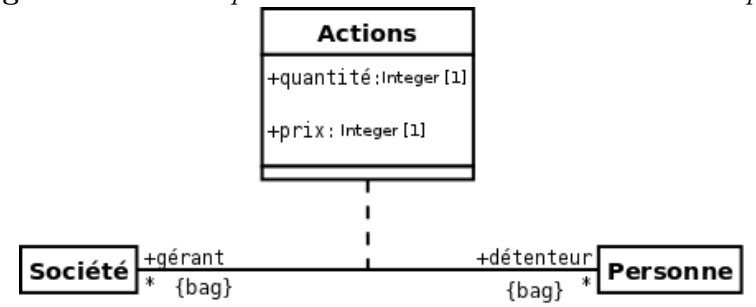
Une classe-association est caractérisée par un trait discontinu entre la classe et l'association qu'elle représente.

Classe-association pour plusieurs associations Il n'est pas possible de rattacher une classe-association à plus d'une association puisque la classe-association constitue elle-même l'association. Dans le cas où plusieurs classe-associations doivent disposer des mêmes caractéristiques, elles doivent hériter d'une même classe possédant ces caractéristiques, ou l'utiliser en tant qu'attribut.

De même, il n'est pas possible de rattacher une instance de la classe d'une classe-association à plusieurs instances de l'association de la classe-association. En effet, la représentation graphique (association reliée par une ligne pointillé à une classe déportée) est trompeuse : une classe-associaiton est une entité sémantique atomique et non composite qui s'intancie donc en bloc.

Liens multiples

Figure 2.29. *Exemple de classe-association avec liens multiples*

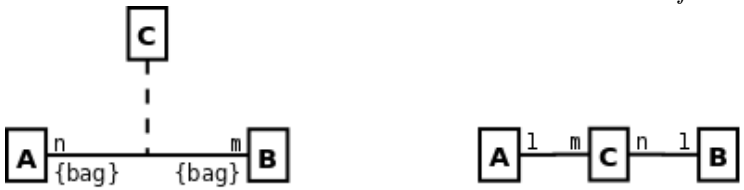


Plusieurs instances d'une même association ne peuvent lier les mêmes objets. Cependant, si l'on s'intéresse à l'exemple de la figure, on voit bien qu'il doit pouvoir y avoir plusieurs instances de la classe-association `Actions` liant une même personne à une même société : une même personne peut acheter à des moments différents des actions d'une même société.

C'est justement la raison de la contrainte bag qui, placée sur les terminaisons d'association de la classe-association `Actions`, indique qu'il peut y avoir des liens multiples impliquant les mêmes paires d'objets.

Équivalences Parfois, l'information véhiculée par une classe-association peut être véhiculée sans perte d'une autre manière.

Figure 2.30. *Deux modélisations modélisant la même information*



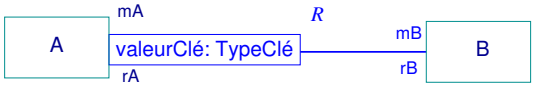
Exercices

ECTS On considère ici une partie d'un système d'information concernant le suivi du cursus de chaque étudiant(e). L'objectif est de connaître les modules que l'étudiant(e) doit suivre et ceux qu'il(elle) a validés.

1. On enregistre l'ensemble des modules que doit suivre l'élève (modules en cours et modules à faire).
2. Un module suivi est, généralement, validé. Dans ce cas, on enregistre le score de l'étudiant(e) : la moyenne qu'il(elle) a eu au module et le « grade » ECTS (A, B...) ; le nombre de crédits est spécifique à chaque module.
3. Le nombre total de crédits qu'a un(e) étudiant(e) est calculable ; il se déduit des informations précédentes.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

2.3.5 Association qualifiée

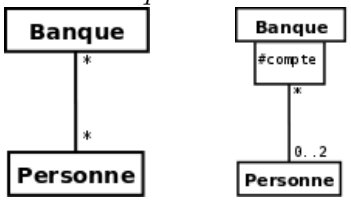


Quand une classe est liée à une autre classe par une association, il est parfois préférable de restreindre la portée de l'association à quelques éléments ciblés (comme un ou plusieurs attributs) de la classe. Ces éléments ciblés sont appelés un qualificatif. Un qualificatif permet donc de sélectionner un ou des objets dans le jeu des objets d'un objet (appelé objet qualifié) relié par une association à un autre objet. L'objet sélectionné par la valeur du qualificatif est appelé objet cible. L'association est appelée association qualifiée. Un qualificatif agit toujours sur une association dont la multiplicité est plusieurs (avant que l'association ne soit qualifiée) du côté cible.

Un objet qualifié et une valeur de qualificatif génèrent un objet cible lié unique. En considérant un objet qualifié, chaque valeur de qualificatif désigne un objet cible unique.

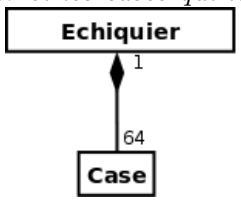
Figure 2.31. *Diagramme représentant l'association entre une banque et ses clients.*

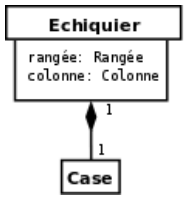
Diagramme équivalent utilisant une association qualifiée.



Un compte dans une banque appartient à au plus deux personnes. Autrement dit, une instance du couple Banque , compte est en association avec zéro à deux instances de la classe Personne. Mais une personne peut posséder plusieurs comptes dans plusieurs banques. C'est-à-dire qu'une instance de la classe Personne peut être associée à plusieurs (zéro compris) instances du couple Banque , compte. Bien entendu, et dans tous les cas, un instance du couple Personne , compte est en association avec une instance unique de la classe Banque.

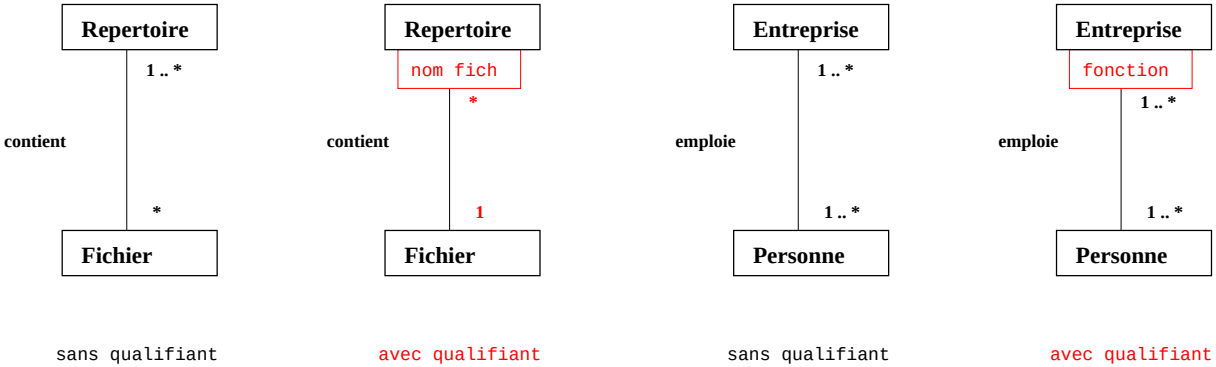
Figure 2.32. *Diagramme représentant l'association entre un échiquier et les cases qui le composent. Diagramme équivalent utilisant une association qualifiée.*





Une instance du triplet Echiquier, rangée, colonne est en association avec une instance unique de la classe Case. Inversement, une instance de la classe Case est en association avec une instance unique du triplet Echiquier, rangée, colonne.

Figure 2.33. *Exemple de la classe Repertoire et de la classe Entreprise*



2.4 Élaboration

Démarche pour bâtir une diagramme de classes :

1. **Trouver les classes du domaine étudié**
   - ▷ En collaboration avec un expert du domaine.
   - ▷ Les classes correspondent généralement à des *concepts* ou des *substantifs* du domaine.
2. **Trouver les associations entre classes**
   - ▷ Les associations correspondent souvent à des verbes, ou des constructions verbales, mettant en relation plusieurs classes, comme ≪ est composé de ≫, ≪ pilote ≫, ≪ travaille pour ≫.
   - ▷ Attention, méfiez vous de certains attributs qui sont en réalité des relations entre classes.
3. **Trouver les attributs des classes**
   - ▷ Les attributs correspondent souvent à des substantifs, ou des groupes nominaux, tels que ≪ la masse d'une voiture ≫ ou ≪ le montant d'une transaction ≫.
   - ▷ Les adjectifs et les valeurs correspondent souvent à des valeurs d'attributs.
4. **Organiser et simplifier le modèle**
   - ▷ En éliminant les classes redondantes et en utilisant l'héritage.
5. **Vérifier les chemins d'accès aux classes**
6. **Itérer et raffiner le modèle**
   - ▷ Un modèle est rarement correct dès sa première construction.

2.5 QCM

2.5.1 Classes avancées

1. La relation entre une classe et l'une de ses interfaces est : _____
   - (a) une relation de dérivation
   - (b) une relation d'agrégation
   - (c) une relation de généralisation
   - (d) une relation de réalisation
   - (e) une relation de spécialisation

2. Les composants communiquent entre eux par _____
   - (a) des objets
   - (b) des appels de méthode
   - (c) des associations
   - (d) des connecteurs
   - (e) des ports et des interfaces

3. Une interface peut être tout sauf : _____
   - (a) réalisée par une classe
   - (b) requise par une classe
   - (c) la généralisation d'une interface
   - (d) la spécialisation d'une interface
   - (e) la généralisation d'une classe

4. Un package peut contenir : _____
   - (a) des classeurs (classifieurs)
   - (b) uniquement d'autres packages
   - (c) uniquement des classes

2.5.2 Associations

1. En UML, une Interface est : _____

   (a) un gestionnaire d'entrée–sortie
   (b) un composant graphique
   (c) un stéréotype de classe
   (d) une classe abstraite

2. Lorsqu'une opération porte la tagged-value {query}, cela signifie que : _____

   (a) l'opération est une demande prioritaire
   (b) l'opération n'est pas définie
   (c) l'opération est une requête à une base de données
   (d) l'opération ne modifie pas l'objet qui est désigné

3. En UML, on représente un constructeur par _____

   (a) une opération avec le stéréotype ≪create≫
   (b) une opération ayant pour nom "create"
   (c) une opération qui porte le même nom que la classe

4. La liste qui suit contient des noms de relation entre classes en UML, à une exception près, laquelle ? _____

   (a) généralisation
   (b) coordination
   (c) contenance
   (d) association
   (e) réalisation

2.5.3 Relations

1. La relation de généralisation entre deux classes est symbolisée par : _____

   (a) un trait discontinu avec une flèche fermée du côté de la sous-classe
   (b) un trait continu avec une flèche ouverte du côté de la super-classe
   (c) un trait continu avec une flèche fermée du côté de la sous-classe
   (d) un trait discontinu avec une flèche ouverte du côté de la super-classe
   (e) un trait continu avec une flèche fermée du côté de la super-classe
   (f) un trait discontinu avec une flèche fermée du côté de la super-classe

2. Une association entre classes est tout sauf : _____

   (a) une composition
   (b) une spécialisation
   (c) une agrégation
   (d) une agrégation composite

3. L'arité d'une association indique : _____

   (a) le nombre de classes en association
   (b) le nombre d'instances en association avec une instance
   (c) le nombre d'instances en association
   (d) la possibilité de la parcourir dans les deux sens

4. Une association à navigation unidirectionnelle est symbolisée par : _____

   (a) un trait continu avec une flèche ouverte
   (b) un trait continu avec une flèche fermée
   (c) un trait continu avec un losange plein
   (d) un trait continu simple
   (e) un trait continu avec un losange vide

3 Diagramme de cas d'utilisation (1 UC)

Sommaire

3.1 Introduction

Exprimer les besoins Comment donner un moyen simple d'exprimer les besoins d'utilisateurs (non informaticiens) ? C'est l'objectif de ce diagramme. En effet, les diagrammes de cas d'utilisation permettent de recenser les grandes fonctionnalités d'un système. Classiquement, un diagramme de cas d'utilisation synthétise la vision utilisateur du système (et non une vision informatique). Il scinde la fonctionnalité du système en unités cohérentes (=les cas d'utilisation). Pour élaborer les cas d'utilisation, il faut utiliser des entretiens avec les utilisateurs. Il s'agit donc de la première étape UML d'analyse d'un système.

Exigences fonctionnelles Ce diagramme est construit en phase de définition des exigences fonctionnelles et enrichi pendant la phase d'analyse en utilisant d'autres modèles (entre-autres les modèles d'interaction). Les objectifs sont :

1. identifier les fonctionnalités du logiciel
2. en définir le périmètre
3. identifier les éléments externes en interaction directe

3.2 Éléments des diagrammes

3.2.1 Acteur

Définition

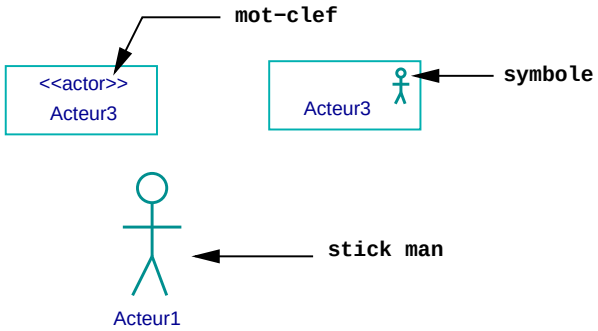
- ▷ rôle joué par une entité externe qui interagit avec le système
- ▷ il peut consulter et/ou modifier l'état du système par messages

Comment les identifier ?

- ▷ utilisateurs humains
- ▷ systèmes connexes qui interagissent également avec le système

Comment les représenter ?

Figure 3.1. *Représentation d'un acteur*



Exemple : Client, Conseiller financier, SI Banque ...

3.2.2 Cas utilisation

Définition

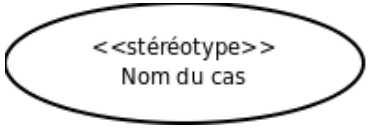
- ▷ Ensemble de séquences d'actions réalisées par le système et qui produisent un résultat observable pour un acteur particulier
- ▷ Spécifie un comportement attendu, sans imposer le mode de réalisation :
  - ◇ il décrit *ce que* le futur devra faire
  - ◇ sans spécifier *comment* il le fera

Comment les identifier ?

- ▷ Ensemble des cas utilisation == exigences fonctionnelles du système
- ▷ Un cas == fonction métier selon le point de vue des acteurs
- ▷ Pour chaque acteur :
  - ◇ rechercher ses utilisations métiers
  - ◇ déterminer dans le cahier des charges les services attendus
    - ▷ Nommez les cas d'utilisation (point de vue acteur) :
      verbe à l'infinitif + complément

Comment les représenter ?

Figure 3.2. *Représentation d'un cas d'utilisation*

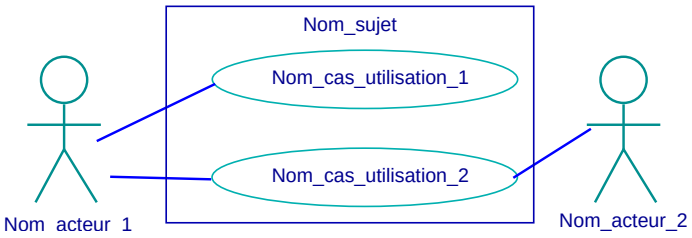


Exemple : Consulter un compte, Retirer de l'argent, Déposer un chèque ...

3.3 Relations entre acteurs et cas d'utilisation

3.3.1 Association

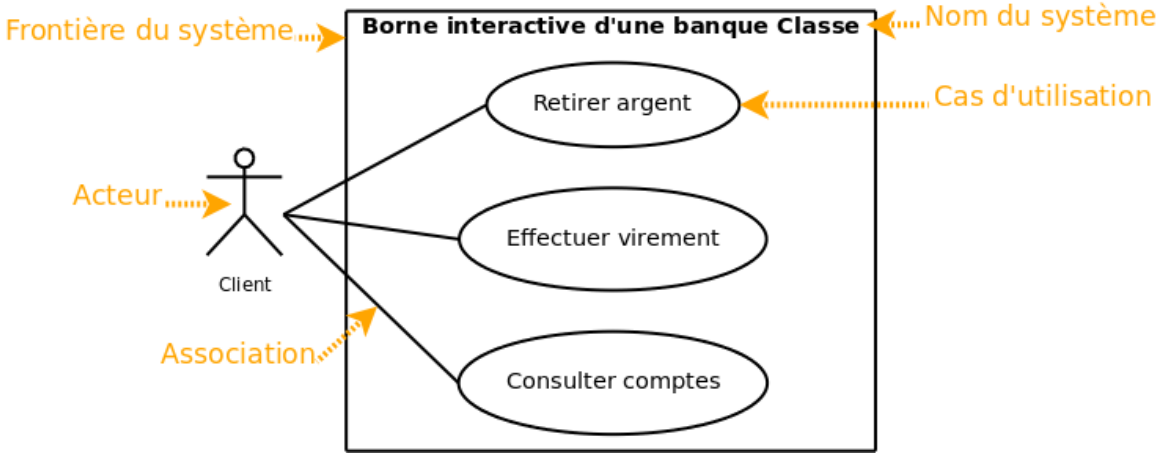
Figure 3.3. *Représentation d'un diagramme de cas d'utilisation*



Un acteur est un élément externe en interaction directe avec le sujet. Un cas d'utilisation est un ensemble fonctionnel cohérent, identifiable extérieurement et fourni par un classeur (le sujet) Une association Acteur – Cas représente le chemin de communication indiquant la participation de l'acteur à la réalisation du cas. Une association est représenté un trait continu.

Le sujet peut être le logiciel lui-même ou, dans le cas de grosses applications, un module du logiciel. La frontière du système est représentée par un cadre. Le nom du système figure à l'intérieur du cadre, en haut. Les acteurs sont à l'extérieur et les cas d'utilisation à l'intérieur.

Exemple 3.1. *Diag. de cas d'utilisation modélisant une borne d'accès à une banque*



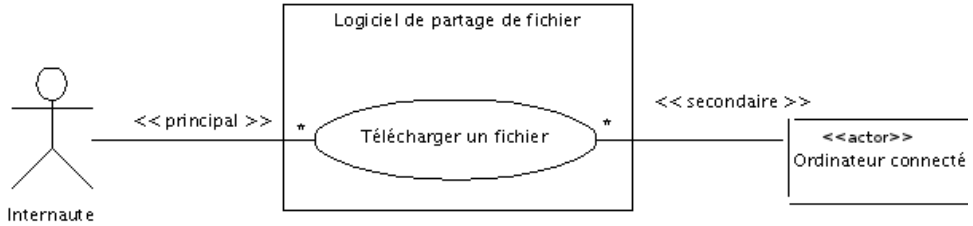
3.3.2 Acteurs principaux et secondaires

Un acteur est qualifié de *principal* pour un cas d'utilisation lorsque ce cas rend service à cet acteur. Les autres acteurs sont alors qualifiés de *secondaires*. Un cas d'utilisation a au plus un acteur

principal. Le stéréotype « primary » vient orner l'association reliant un cas d'utilisation à son acteur principal. Le stéréotype « secondary » est utilisé pour les acteurs secondaires.

Un acteur principal obtient un résultat observable du système tandis qu'un acteur secondaire est sollicité pour des informations complémentaires. En général, l'acteur principal initie le cas d'utilisation par ses sollicitations.

Exemple 3.2. *Diag. de cas d'utilisation représentant un logiciel de partage de fichiers*



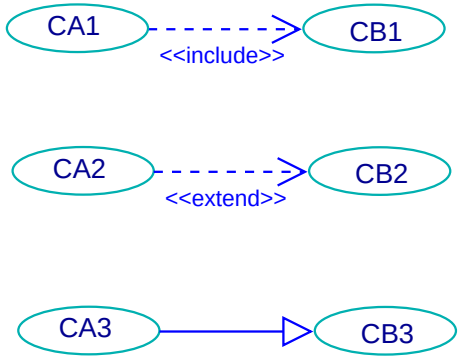
3.4 Relations entre cas d'utilisation

Il existe trois types de relations :

1. l'*inclusion* : `CA1` inclut `CB1`
   1 réalisation de `CA1` ⇒ **1** réalisation de `CB1`
2. l'*extension* : `CA2` étend `CB2` dans un certain contexte
   Selon le contexte, on réalise soit `CA2`, soit `CB2`
3. la *spécialisation* : `CA3` spécialise `CB3`
   Selon le contexte, on réalise soit `CA3`, soit `CB3`

Une inclusion ou une extension se représente par une flèche avec un trait pointillé. Si le cas A inclut ou étend le cas B, la flèche est dirigée de A vers B. Le symbole utilisé pour la spécialisation est un flèche avec un trait pleins dont la pointe est un triangle fermé désignant le cas le plus général.

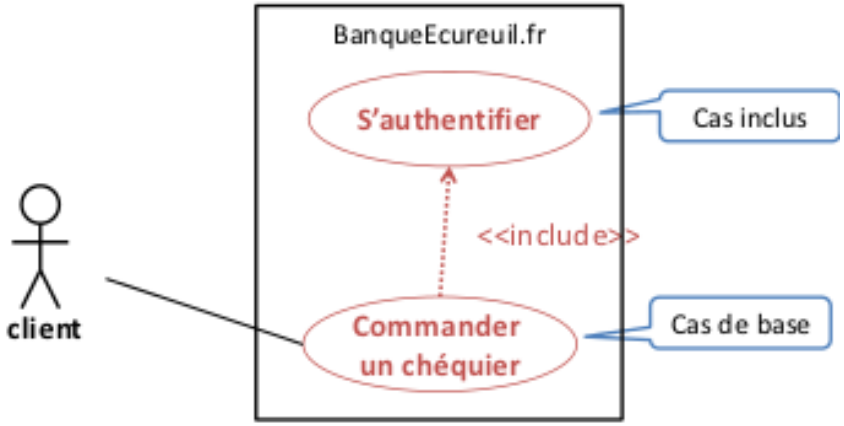
Figure 3.4. *Types et représentation*



3.4.1 Inclusion

Un cas A inclut un cas B si le comportement décrit par le cas A inclut le comportement du cas B (le cas A dépend de B). Lorsque A est sollicité, B l'est obligatoirement, comme une partie de A. Cette dépendance est symbolisée par le stéréotype ≪ include ≫.

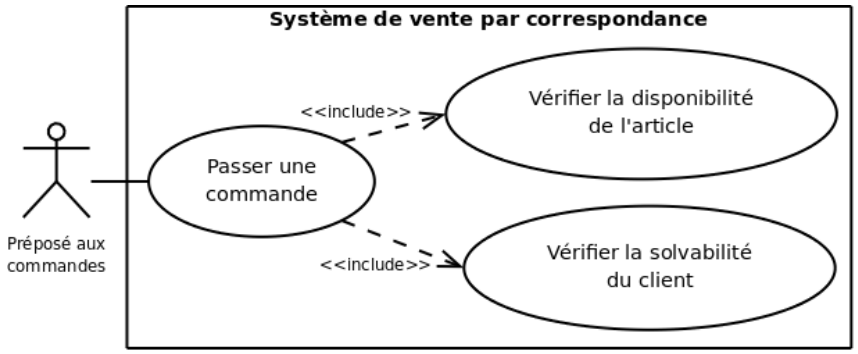
Exemple 3.3. *Le cas inclus est ajouté obligatoirement au cas de base*



Les inclusions permettent essentiellement de factoriser une partie de la description d'un cas d'utilisation qui serait commune à d'autres cas d'utilisation. Les inclusions permettent également de

décomposer un cas complexe en sous-cas plus simples. Cependant, il ne faut surtout pas abuser de ce type de décomposition : il faut éviter de réaliser du découpage fonctionnel d'un cas d'utilisation en plusieurs sous-cas d'utilisation pour ne pas retomber dans le travers de la décomposition fonctionnelle.

Exemple 3.4. *Identifier une partie commune aux différents cas d'utilisation et de la factoriser dans un nouveau cas inclus dans ces derniers.*

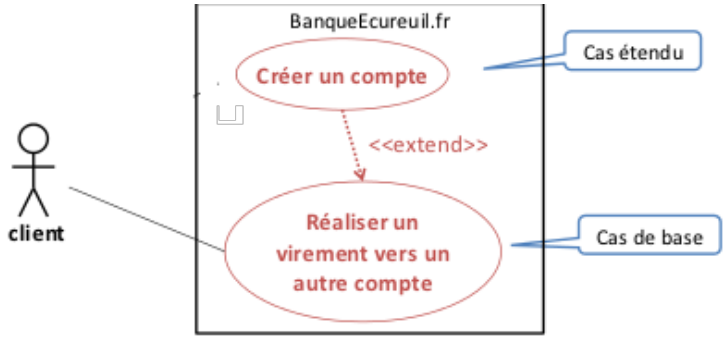


Attention les cas d'utilisation ne s'enchaînent pas (aucune représentation temporelle).

3.4.2 Extension

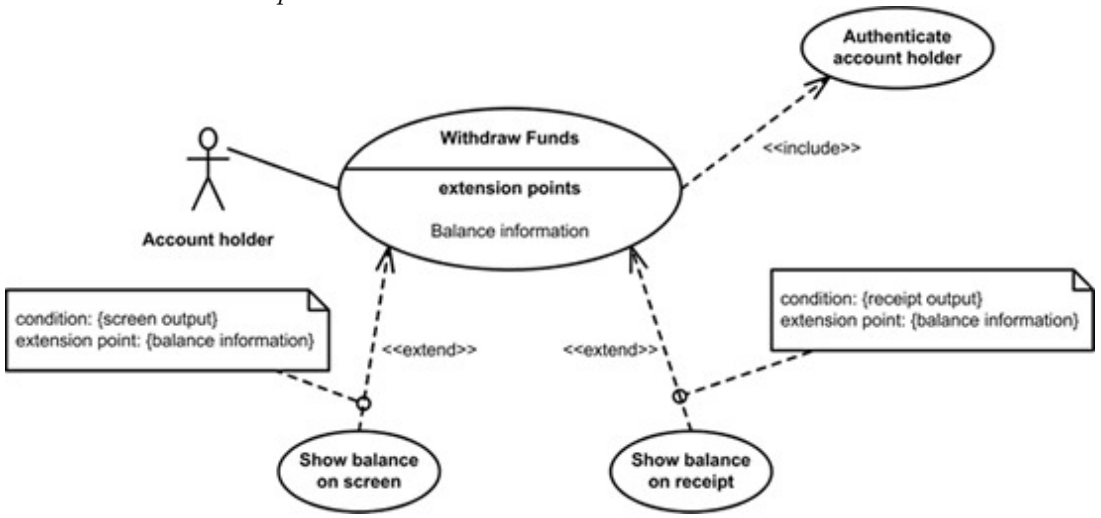
La relation d'extension a une sémantique qui a un sens du point de vue métier (au contraire des deux autres). On dit qu'un cas d'utilisation A étend un cas d'utilisation B lorsque le cas d'utilisation A peut être appelé au cours de l'exécution du cas d'utilisation B. Exécuter B peut éventuellement entraîner l'exécution de A : contrairement à l'inclusion, l'extension est optionnelle. Cette dépendance est symbolisée par le stéréotype ≪ extend ≫.

Exemple 3.5. *Enrichir un cas d'utilisation par un autre, cependant, cet enrichissement est optionnel.*



L'extension peut intervenir à un point précis du cas étendu. Ce point s'appelle le point d'extension. Il porte un nom, qui figure dans un compartiment du cas étendu sous la rubrique point d'extension, et est éventuellement associé à une contrainte indiquant le moment où l'extension intervient. Une extension est souvent soumise à condition. Graphiquement, la condition est exprimée sous la forme d'une note.

Exemple 3.6. *L'extension se fait dans le cas d'utilisation de base, en un point précis appelé point d'extension*

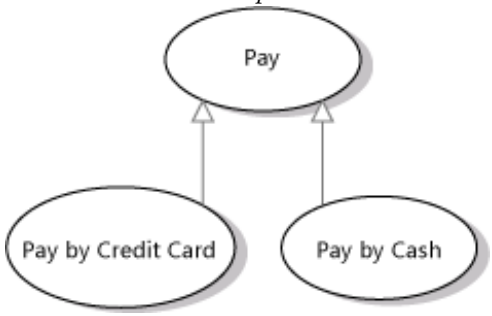


Exercice 3.1. *Supermarché*
Pour acheter un produit, un client régulier peut éventuellement présenter sa carte de réduction sous réserve qu'il la possède sur lui. Formalisez ces élements dans un diag. de cas d'utilisation.

3.4.3 Généralisation/Spécialisation

Un cas A est une généralisation d'un cas B si B est un cas particulier de A. Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages orientés objet.

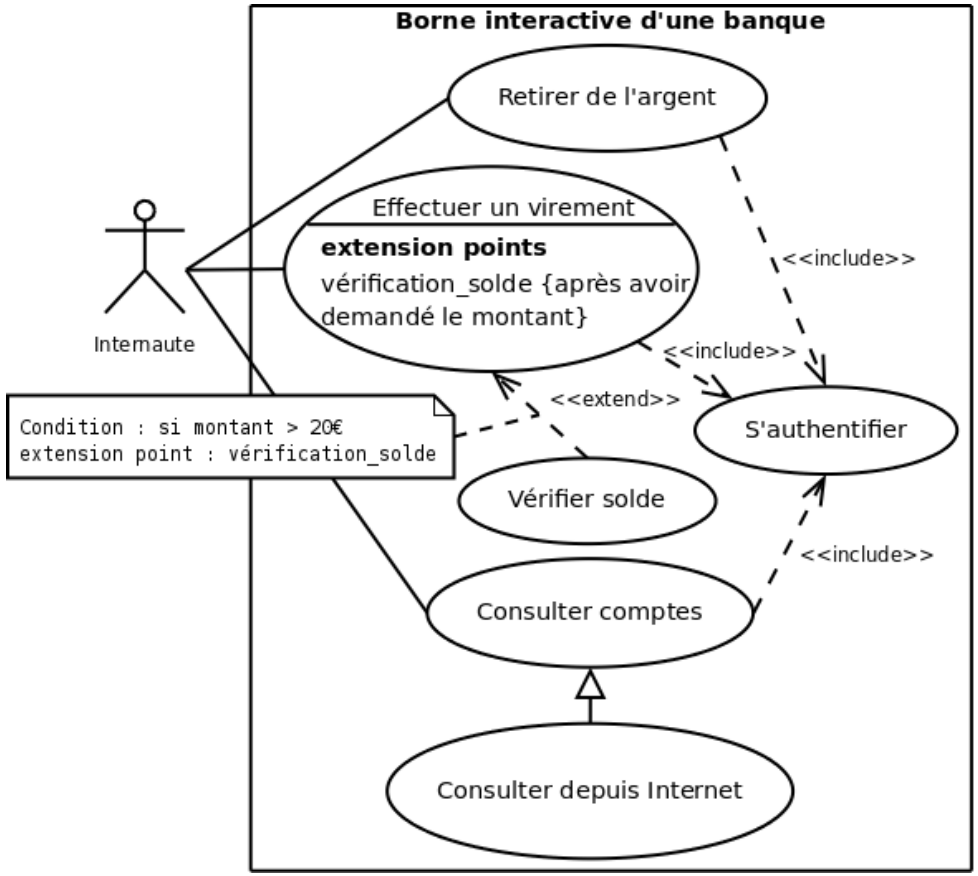
Exemple 3.7. *Formaliser les variations importantes sur le même cas d'utilisation*



Une relation de généralisation/spécialisation permet d'exprimer que les cas d'utilisation descendants héritent de la description de leur parent commun. Ils peuvent cependant comprendre chacune des interactions spécifiques supplémentaires, ou modifier les interactions dont ils ont hérités.

3.4.4 Exemple complet

Exemple 3.8. *Exemple de diagramme de cas d'utilisation*



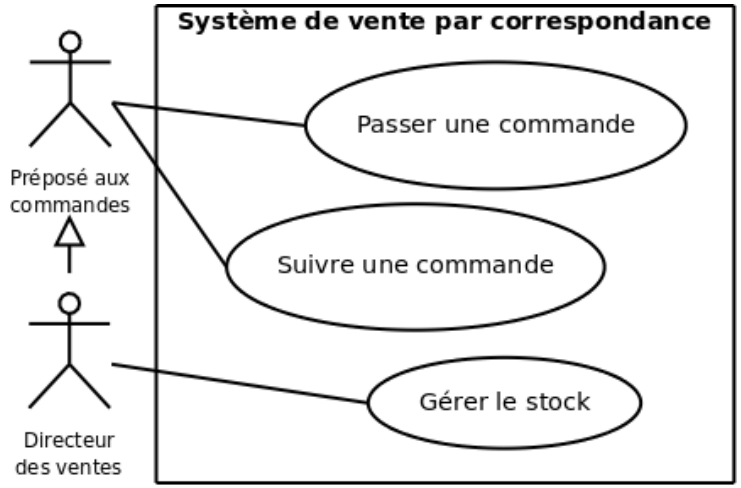
La consultation d'un compte via Internet est un cas particulier de la consultation. La vérification du solde du compte n'intervient que si la demande de retrait dépasse 20 euros.

3.5 Relations entre acteurs

La seule relation possible entre deux acteurs est la généralisation : un acteur A est une généralisation d'un acteur B si l'acteur A peut être substitué par l'acteur B. Dans ce cas, tous les cas d'utilisation accessibles à A le sont aussi à B, mais l'inverse n'est pas vrai.

Le symbole utilisé pour la généralisation entre acteurs est une flèche avec un trait plein dont la pointe est un triangle fermé désignant l'acteur le plus général.

Figure 3.5. *Relations entre acteurs*



Le directeur des ventes est un préposé aux commandes avec un pouvoir supplémentaire : en plus de pouvoir passer et suivre une commande, il peut gérer le stock. Par contre, le préposé aux commandes ne peut pas gérer le stock.

3.6 Scénarios : description textuelle

Une fois les cas d'utilisation identifiés, il faut les décrire. Cette description repose sur la notion de scénario.

Figure 3.6. *Chaque cas d'utilisation est décrit par des scénarios*

Définition Un scénario est une succession particulière d'enchaînements s'exécutant du début à la fin du cas d'utilisation.

Un cas d'utilisation sera représenté par :

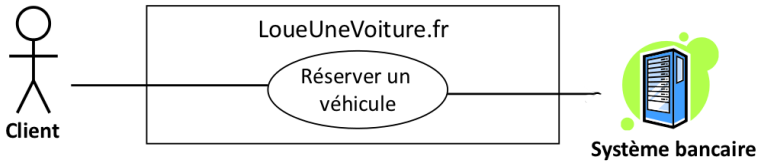
- ▷ un scénario nominal
- ▷ plusieurs scénarios alternatifs (qui se terminent normalement)
- ▷ plusieurs scénarios d'erreur (qui se terminent par un échec)

Fiche La fiche de description textuelle d'un cas d'utilisation n'est pas normalisée... Cependant, on peut utiliser la structuration suivante :

Sommaire d'identification (obligatoire)	Inclut titre, résumé, dates de création et de modification, version, responsable, acteurs ...
Description des scénarios (obligatoire)	Décrit le scénario nominal, les scénarios d'erreur, les pré/post-conditions
Exigences non-fonctionnelles (optionel)	Ajoute, si c'est pertinent, les informations suivantes : fréquence ; disponibilté, fiabilité, confidentialité ...

Exemple scénario

Exemple 3.9. *Réserver un véhicule : ce cas d'utilisation permet à un client internaute de saisir une demande de réservation.*



Exercice 3.2. *Boutique Web Pour acheter un produit, le scénario peut être le suivant :*
Le client parcourt le catalogue et ajoute les articles désirés à son panier éléctronique. Quand il veut payer, il fournit les informations sur la livraison et sur la carte de crédit et confirme l'achat. Le système vérifie que la carte de crédit est autorisée, et confirme la vente iméédiatement puis par e-mail. Formalisez ce scénario par un tableau. Ajouter la possibilité que l'uatorisation peut échouer.

Sommaire	Scénario nominal ≪ Réserver un véhicule ≫
Description	1. Le client saisit son code et son login d'identification 2. Le système vérifie le code et le login d'identification 3. Le système demande au client de saisir les informations sur la réservation 4. Le client saisit les informations sur la réservation 5. Le système interroge l'acteur système bancaire pour vérifier l'acompte 6. Le système bancaire donne une réponse favorable 7. Le système envoie au client, un message de confirmation de la demande

Sommaire	Scénario alternatif ≪ Réserver un véhicule ≫
Description	SA1 : code d'identification erroné pour la première ou la deuxième fois SA1 démarre au point 2 du scénario nominal 3. Le système indique au client que le code est erroné, pour la première ou la deuxième fois. Le scénario nominal reprend au point 1.

Sommaire	Scénario d'erreur ≪ Réserver un véhicule ≫
Description	SE1 : code d'identification erroné pour la troisième fois SE1 démarre au point 2 du scénario nominal 3. Le système indique au client que le code est erroné pour la troisième fois. Le cas d'utilisation se termine en échec (l'objectif n'est pas atteint).

3.7 Utilisation

Quand utiliser les cas d'utilisation ?

- ▷ En phase d'élaboration
- ▷ En discutant avec les utilisateurs

Remarque : Un projet de 10 années-hommes devrait comprendre environ 12 cas d'utilisation

3.8 QCM

1. Un acteur est : _____

   (a) une instance de classe qui réalise une fonctionnalité
   (b) un déclencheur d'une opération d'un classifier
   (c) un élément externe qui participe à la réalisation d'un cas d'utilisation
   (d) une classe active du système

2. La relation entre deux cas d'utilisation peut être tout sauf : _____

   (a) une inclusion
   (b) une réalisation
   (c) une généralisation
   (d) une extension

3. Les relations entre acteurs et cas d'utilisation sont des : _____

   (a) extensions
   (b) réalisations
   (c) généralisations
   (d) inclusions
   (e) associations

4. L'extension d'un cas d'utilisation permet d'exprimer : _____

   (a) la durée de la réalisation d'un cas d'utilisation
   (b) un élément commun à plusieurs cas d'utilisation
   (c) le traitement d'un cas particulier
   (d) un cas d'utilisation abstrait
   (e) les conditions de réalisation d'un cas d'utilisation

4 Diagrammes d'interaction (2 UC)

Sommaire

4.1 Introduction

4.1.1 Formaliser les scénarios des cas d'utilisation

Une fois que les cas d'utilisation et les scénarios associés sont définis, il s'agit de les formaliser.

Figure 4.1. *Du diag. de cas d'utilisation au diag. d'interactions*

$\boxed{\textit{diag. cas d'utilisation}} \rightarrow \boxed{\textit{scénarios}}$

$\boxed{\textit{scénario}} \rightarrow \boxed{\textit{diag. seq/comm}}$

Pour cela, il faut définir les interactions entre les *objets* (instances) qui participent au scénario. Les diagrammes d'interaction permettent d'établir un lien entre les diagrammes de cas d'utilisation et les diagrammes de classes : ils montrent comment des objets (i.e. des instances de classes) communiquent pour réaliser une certaine fonctionnalité. Ils apportent ainsi un aspect dynamique à la modélisation du système.

4.1.2 Quel type de diagramme ?

Prenons l'exemple du cas d'utilisation *Piloter*. Le scénario nominal pourrait être : un pilote démarre une voiture ce qui allume un moteur. Comment formaliser les communications entre instances (démarrer, allumer) ? → diag. de communication.

Comment formaliser le séquencement des interactions (1 : démarrer ; 2 : allumer) ? → diag. séquence.

Avant cela, il faut représenter les instances (objets) (un pilote, une voiture, un moteur)

4.1.3 Objet : instance de classifier

Une instance est représentée sous la forme suivante :

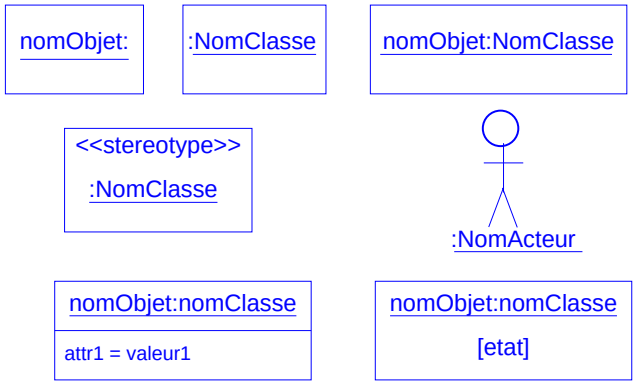
nom d'instance : nom de classe

Exemple 4.1. *Exemples d'objet*

jean : Personne

pierre : Personne

Figure 4.2. *Représentation d'un objet*



Exercice 4.1. *Cas d'utilisation "Piloter"*
Représentez les objets associés.

Exercice 4.2. *Cas d'utilisation "Vente Immobilière"*
Représentez les objets associés. Dans ce cas, un notaire établit un contrat entre deux personnes.

4.2 Diagramme de séquence

Les principales informations contenues dans un diagramme de séquence sont les messages échangés entre les lignes de vie, présentés dans un ordre chronologique. Ainsi, contrairement au diagramme de communication, le temps y est représenté explicitement par une dimension (la dimension verticale) et s'écoule de haut en bas

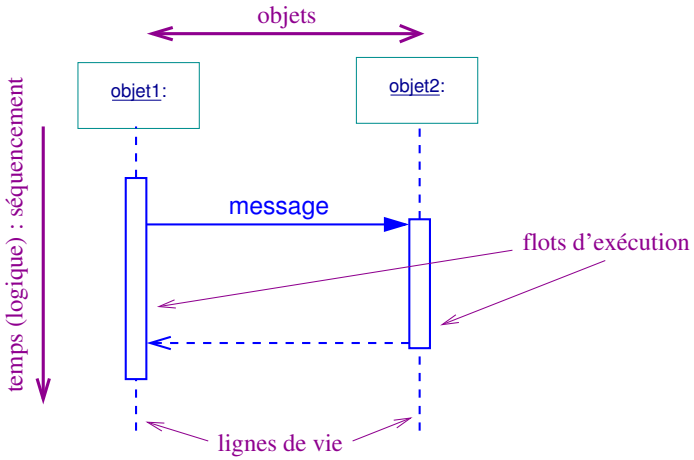


Figure 4.3.

4.2.1 Lignes de vie

Une ligne de vie se représente par un rectangle, auquel est accroché une ligne verticale pointillée, contenant une étiquette dont la syntaxe est :
`[<nom_du_role>] : [<Nom_du_type>]`

Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

4.2.2 Messages

Un message définit une communication particulière entre des lignes de vie. Plusieurs types de messages existent, les plus commun sont :

- ▷ l'envoi d'un signal ;
- ▷ l'invocation d'une opération ;
- ▷ la création ou la destruction d'une instance.

Messages asynchrones Une interruption ou un évènement sont de bons exemples de signaux. Ils n'attendent pas de réponse et ne bloquent pas l'émetteur qui ne sait pas si le message arrivera à destination, le cas échéant quand il arrivera et s'il serra traité par le destinataire. Un signal est, par définition, un message asynchrone.

Graphiquement, un message asynchrone se représente par une flèche en traits pleins et à l'extrémité ouverte partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible

Figure 4.4.

Messages synchrones L'émetteur reste alors bloqué le temps que dure l'invocation de l'opération.

Graphiquement, un message synchrone se représente par une flèche en traits pleins et à l'extrémité pleine partant de la ligne de vie d'un objet expéditeur et allant vers celle de l'objet cible. Ce message peut être suivi d'une réponse qui se représente par une flèche en pointillé.

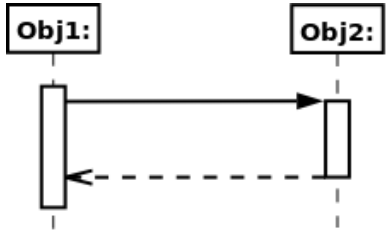


Figure 4.5.

Messages de création et destruction d'instance La création d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie.

La destruction d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet. La destruction d'un objet n'est pas nécessairement consécutive à la réception d'un message.

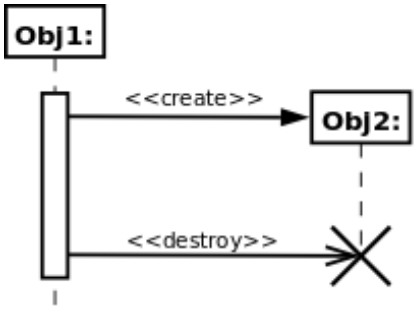


Figure 4.6.

Événements et messages UML permet de séparer clairement l'envoi du message, sa réception, ainsi que le début de l'exécution de la réaction et sa fin.

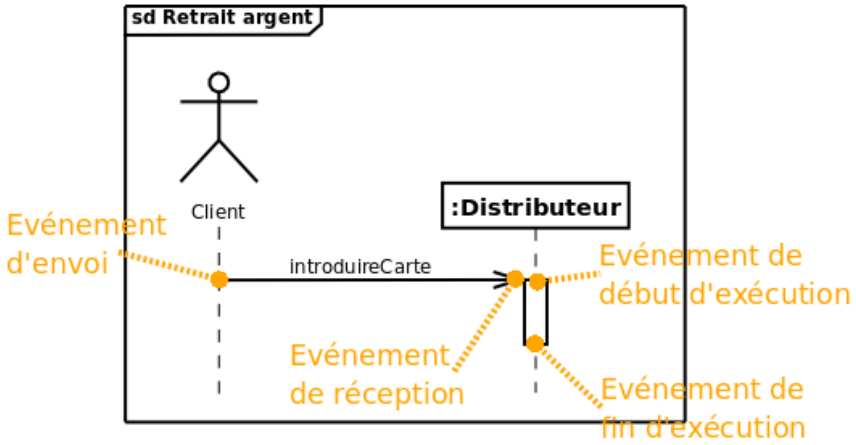


Figure 4.7.

Syntaxe des messages et des réponses Dans la plupart des cas, la réception d'un message est suivie de l'exécution d'une méthode d'une classe. Cette méthode peut recevoir des arguments et la syntaxe des messages permet de transmettre ces arguments. La syntaxe de ces messages est la même que pour un diagramme de communication excepté deux points :

- ▷ la direction du message est directement spécifiée par la direction de la flèche qui matérialise le message, et non par une flèche supplémentaire au dessus du connecteur reliant les objets comme c'est le cas dans un diagramme de communication ;
- ▷ les numéros de séquence sont généralement omis puisque l'ordre relatif des messages est déjà matérialisé par l'axe vertical qui représente l'écoulement du temps.

La syntaxe de réponse à un message est la suivante :

`[<attribut> = ] message [ : <valeur_de_retour>]`

où `message` représente le message d'envoi.

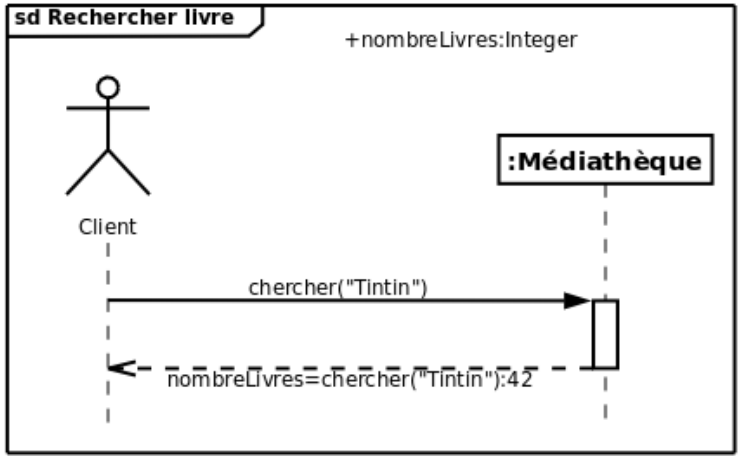


Figure 4.8.

Message perdu et trouvé Un message perdu est tel que l'événement d'envoi est connu, mais pas l'événement de réception. Il se représente par une flèche qui pointe sur une petite boule noire.

Un message trouvé est tel que l'événement de réception est connu, mais pas l'événement d'émission. Une flèche partant d'une petite boule noire représente un message trouvé.

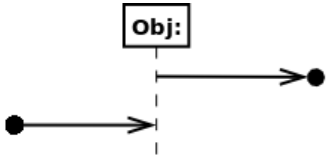


Figure 4.9.

Exécution de méthode et objet actif Un objet actif initie et contrôle le flux d'activités. Graphiquement, la ligne pointillée verticale d'un objet actif est remplacée par un double trait vertical.

Un objet passif, au contraire, a besoin qu'on lui donne le flux d'activité pour pouvoir exécuter une méthode. La spécification de l'exécution d'une réaction sur un objet passif se représente par un rectangle blanc ou gris placé sur la ligne de vie en pointillée. Le rectangle peut éventuellement porter un label.

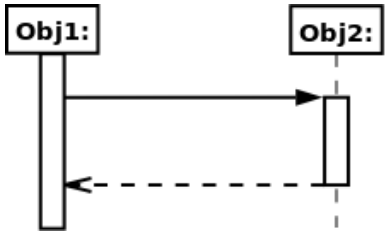


Figure 4.10.

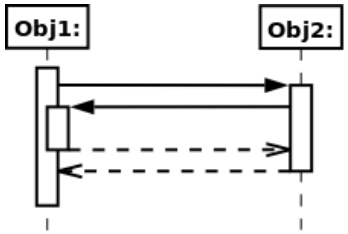


Figure 4.11.

4.2.3 Fragements

Présentation Un fragment combiné représente des articulations d'interactions. Il est défini par un opérateur et des opérandes. L'opérateur conditionne la signification du fragment combiné. Il existe 12 d'opérateurs définis dans la notation UML 2.0. Les fragments combinés permettent de décrire des diagrammes de séquence de manière compacte. Les fragments combinés peuvent faire intervenir l'ensemble des entités participant au scénario ou juste un sous-ensemble.

Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté dans un rectangle dont le coin supérieur gauche contient un pentagone. Dans le pentagone figure le type de la combinaison, appelé opérateur d'interaction. Les opérandes d'un opérateur d'interaction sont séparés par une ligne pointillée. Les conditions de choix des opérandes sont données par des expressions booléennes entre crochets ([ ]).

La liste suivante regroupe les opérateurs d'interaction par fonctions :

- ▷ choix et boucle : **alt**ernative, **opt**ion, **break**, **loop**
- ▷ contrôle d'envoi en parallèle de messages : **par**allel, **critical region**
- ▷ contrôle d'envoi de messages : **ignore**, **consider**, **assert**ion, **negative**
- ▷ ordre d'envoi des messages : **weak** sequencing, **strict** sequencing

Alt

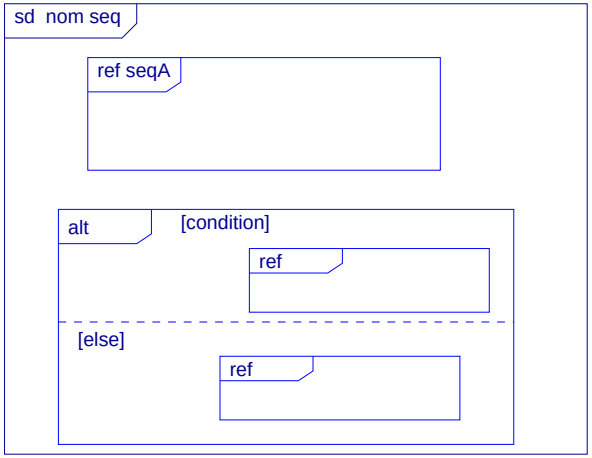
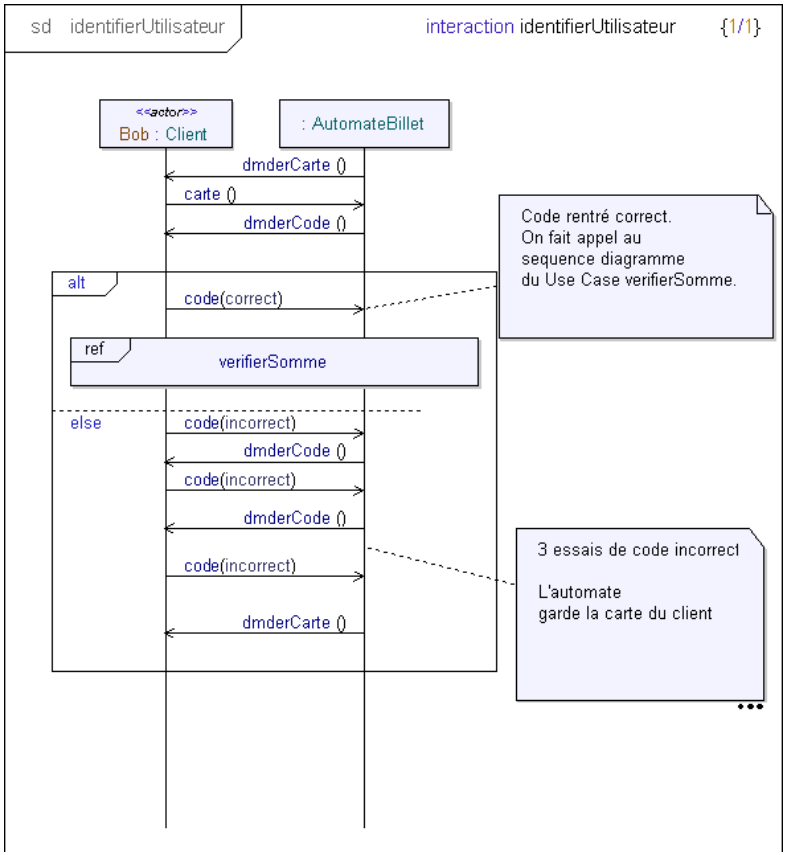


Figure 4.12.

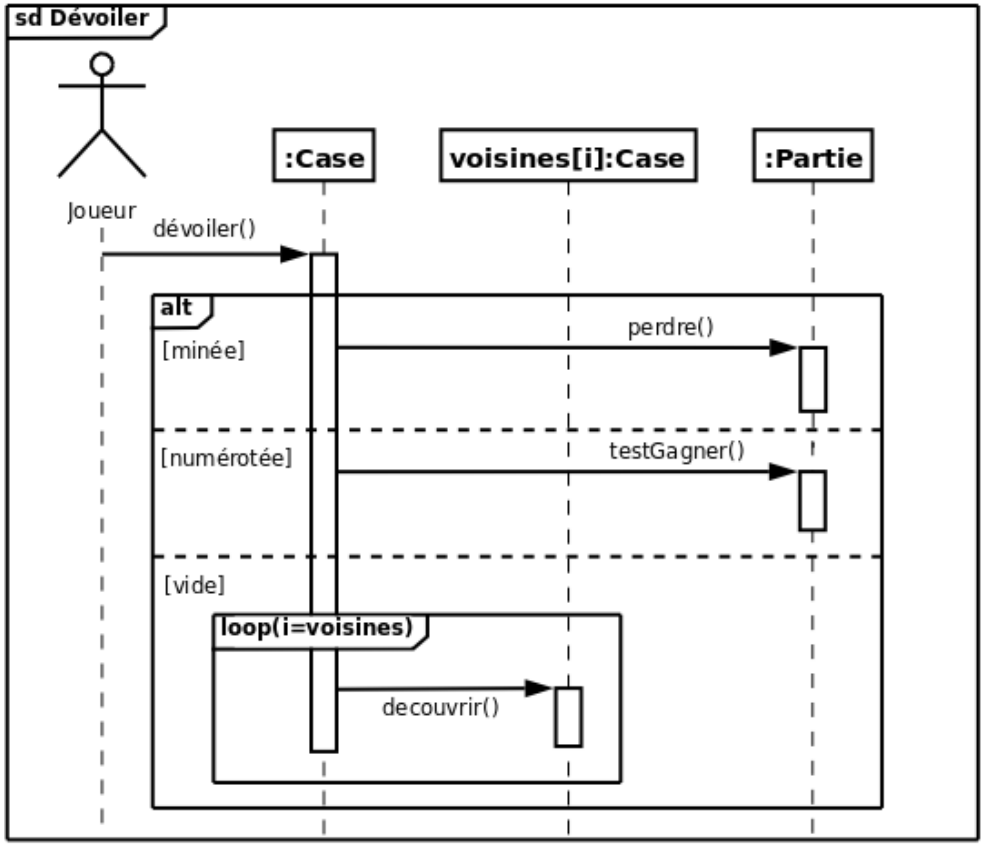
L'opérateur `alt` désigne un choix, une alternative. Il représente deux comportements possibles : c'est en quelque sorte l'équivalent du SI...ALORS...SINON : donc, une seule des deux branches sera réalisée dans un scénario donné. La condition d'exécution d'une des deux branches (l'équivalent du SI) peut être explicite ou implicite. L'utilisation de l'opérateur `else` permet d'indiquer que la branche est exécutée si la condition du `alt` est fausse.

Exemple 4.3. *Alternative*



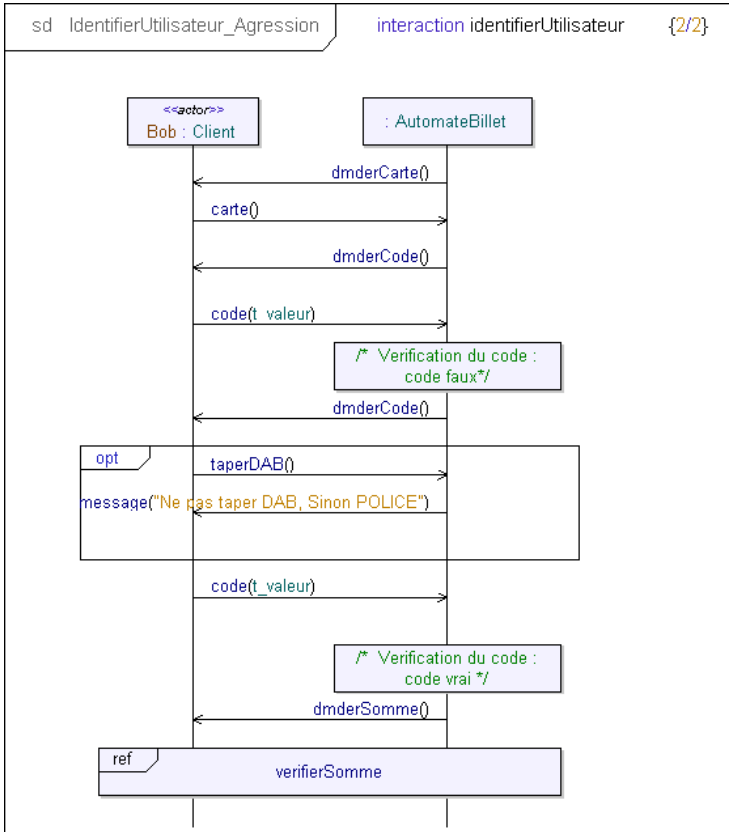
- ▷ soit l'utilisateur rentre un code correct et dans ce cas le diagramme de séquence relatif à la vérification du code est appelé,
- ▷ soit l'utilisateur rentre un code erroné, trois fois, et sa carte est gardée (le distributeur se ré-initialise et demande à nouveau une carte).

Exemple 4.4. *Alternative : jeu du démineur*


sd Dévoiler
Joueur
:Case
voisines[i]:Case
:Partie
dévoiler()
alt
[minée]
perdre()
[numérotée]
testGagner()
[vide]
loop(i=voisines)
decouvrir()

Opt L'opérateur `opt` désigne un fragment combiné optionnel comme son nom l'indique : c'est à dire qu'il représente un comportement qui peut se produire... ou pas. Un fragment optionnel est équivalent à un fragment `alt` qui ne posséderait pas d'opérande `else` (qui n'aurait qu'une seule branche). Un fragment optionnel est donc une sorte de SI...ALORS.

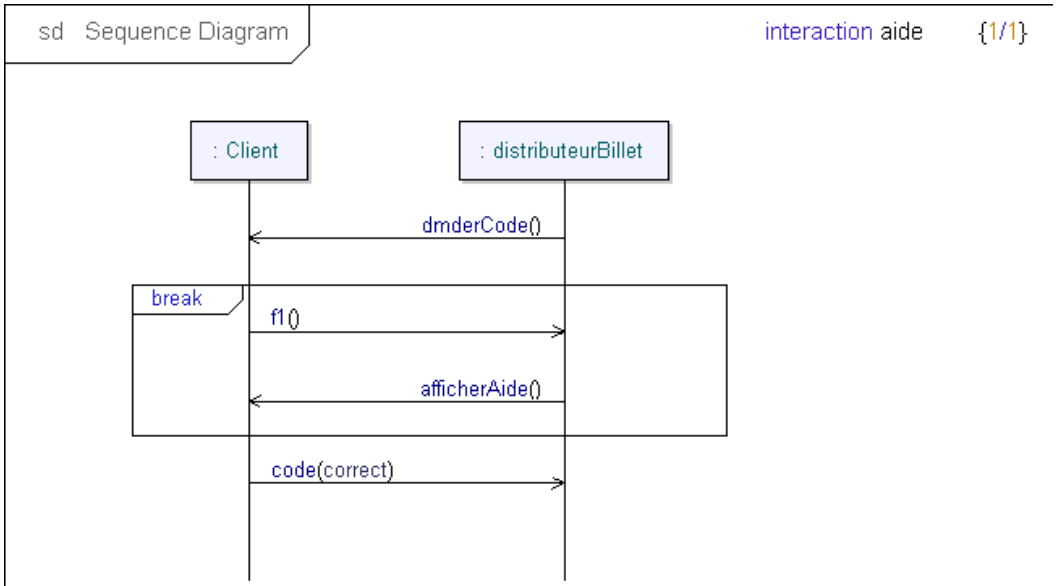
Exemple : L'utilisateur, si il est mécontent, peut se défouler sur le distributeur de billets. En revanche, la plupart des utilisateurs contiennent leur agressivité et restent corrects envers le distributeur de billet. L'opérateur `opt` montre cette possibilité.



Exemple 4.5.

Break L'opérateur `break` est utilisé dans les fragments combinés qui représentent des scenarii d'exception en quelque sorte. Les intéractions de ce fragment seront exécutées à la place des intéractions décrites en dessous. Il y a donc une notion d'interruption du flot "normal" des intéractions.

Exemple : L'utilisateur, lorsque le distributeur lui demande son code, peut choisir de rentrer son code ou de consulter l'aide. Si il choisit de consulter l'aide, le flot d'intéraction relatif à la saisie du code est interrompu. Les intéractions de l'opérateur `break` sont "exécutées".



Exemple 4.6.

Loop L'opérateur `Loop` (boucle) est utilisé pour décrire un ensemble d'intéraction qui s'exécutent en boucle. En général, une contrainte appelée garde indique le nombre de répétitions (minimum et maximum) ou bien une condition booléenne à respecter.

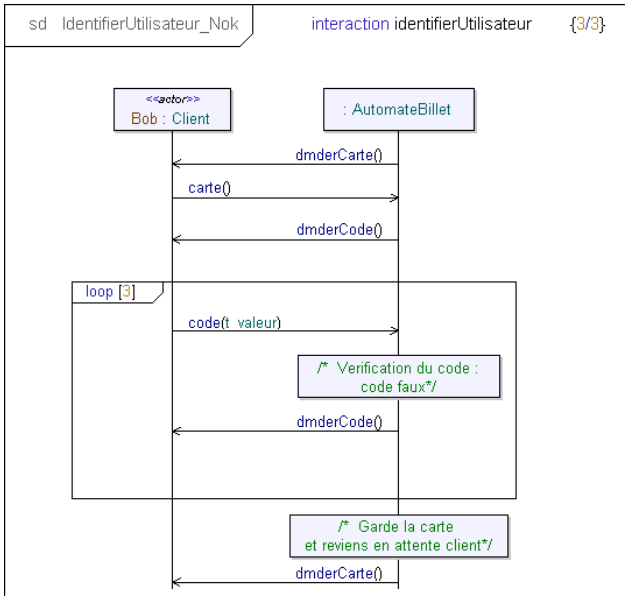
La syntaxe de la boucle est la suivante :

```
loop[ '('<minInt> [ ',' <maxInt> ] ')' ]
```

La condition de garde est placée entre crochets sur la ligne de vie. La boucle est répétée au moins minInt fois avant qu'une éventuelle condition de garde booléenne ne soit testée. Tant que la condition est vraie, la boucle continue, au plus maxInt fois. Cette syntaxe peut être remplacée par une indication intelligible.

Exemple : Le diagramme de séquence indique que lorsque l'utilisateur se trompe trois fois de code, la carte est gardée et le distributeur se remet en mode d'attente d'une carte.

Exemple 4.7.



Par/Seq

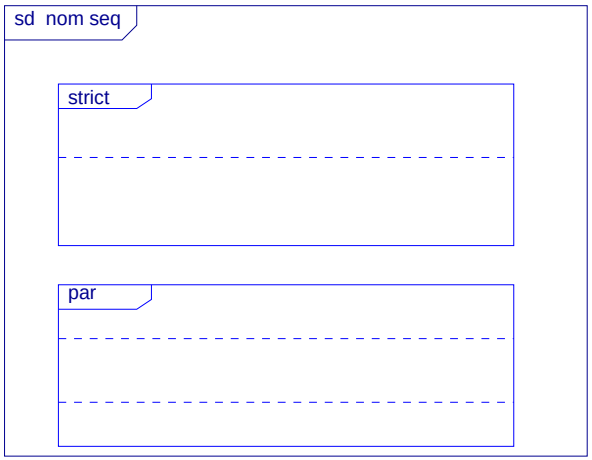
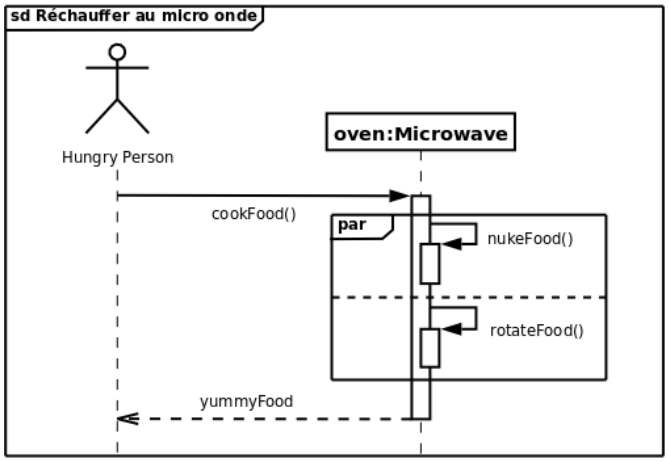
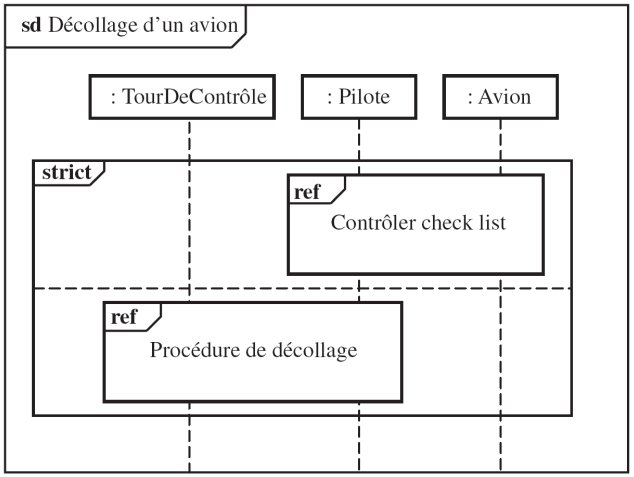


Figure 4.13.

- ▷ Un fragments combiné de type `parallel`, ou `par`, possède au moins deux sous-fragments exécutés simultanément
- ▷ Un fragments combiné de type `strict sequencing`, ou `strict`, possède au moins deux sous-fragments. Ceux-ci s'exécutent selon leur ordre d'apparition au sein du fragment combiné. Ce fragment combiné est utile surtout lorsque deux parties d'un diagramme n'ont pas de ligne de vie en commun.



Exemple 4.8.

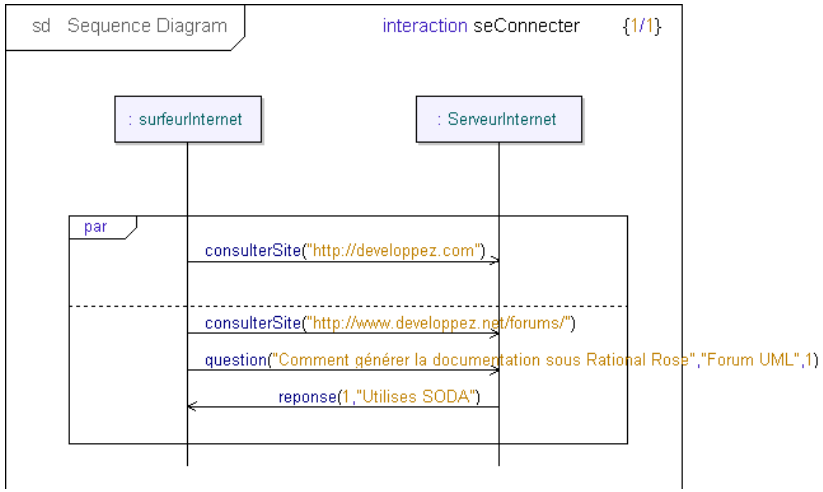


Exemple 4.9.

L'opérateur `par` est utilisé pour représenter des intéractions ayant lieu en parallèle. Les intéractions des différents opérandes (les deux branches de notre opérateur ci-dessous) peuvent donc se mélanger, s'intercaler, dans la mesure où l'ordre imposé dans chaque opérande est respecté.

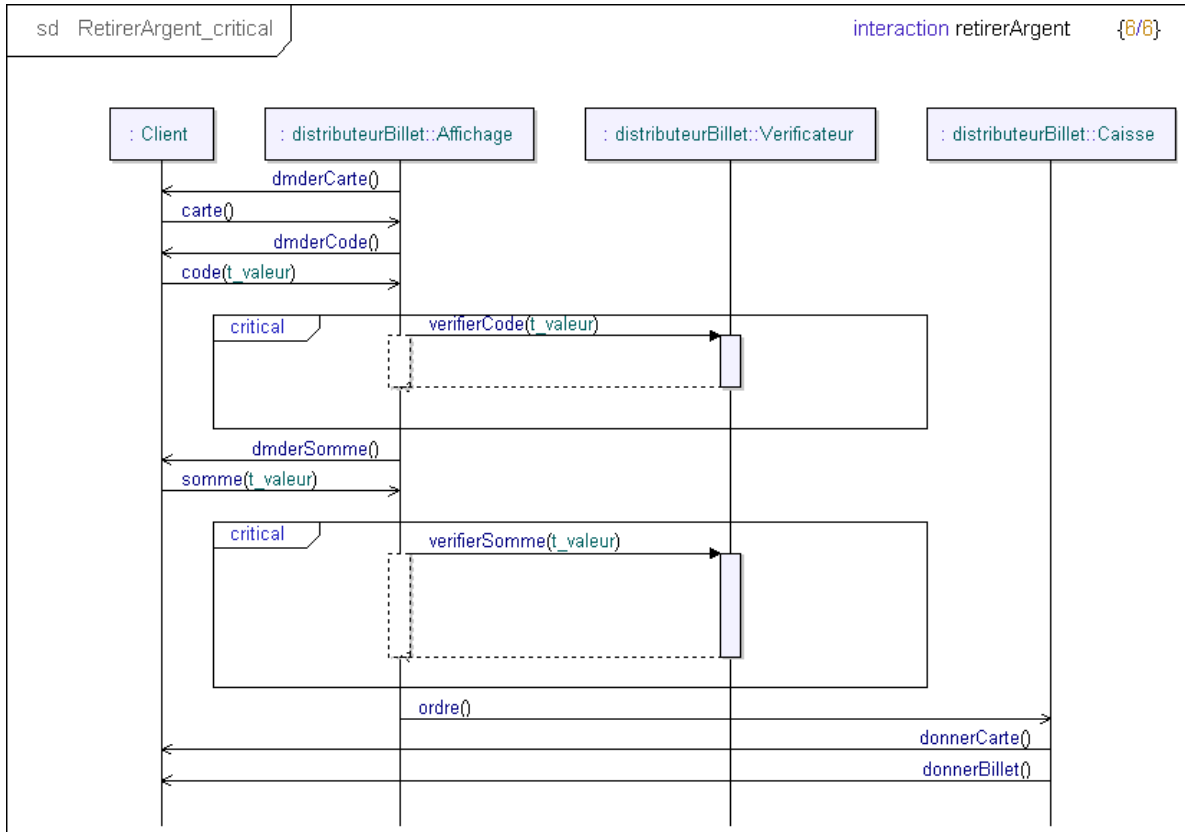
Exemple : Un développeur averti ayant accès à Internet peut consulter en parallèle, soit le site http ://www.developpez.com soit le site http ://www.developpez.net/forums/ sans préférence d'ordre (il peut commencer par consulter les forums puis les cours, soit l'inverse).

Exemple 4.10.


sd Sequence Diagram
interaction seConnecter {1/1}
: surfeurInternet
: ServeurInternet
par
consulterSite("http://developpez.com")
consulterSite("http://www.developpez.net/forums/")
question("Comment générer la documentation sous Rational Rose","Forum UML",1)
reponse(1,"Utilises SODA")

Critical L'opérateur `critical` (critique) désigne une section critique. Une section critique permet d'indiquer que les intéractions décrites dans cet opérateur ne peuvent pas être interrompues par d'autres intéractions décrites dans le diagramme. On dit que l'opérateur impose un traitement atomique des intéractions qu'il contient.

Exemple : On ne souhaite pas que l'utilisateur puisse obtenir des billets avec un code erroné et une somme demandée incorrecte.

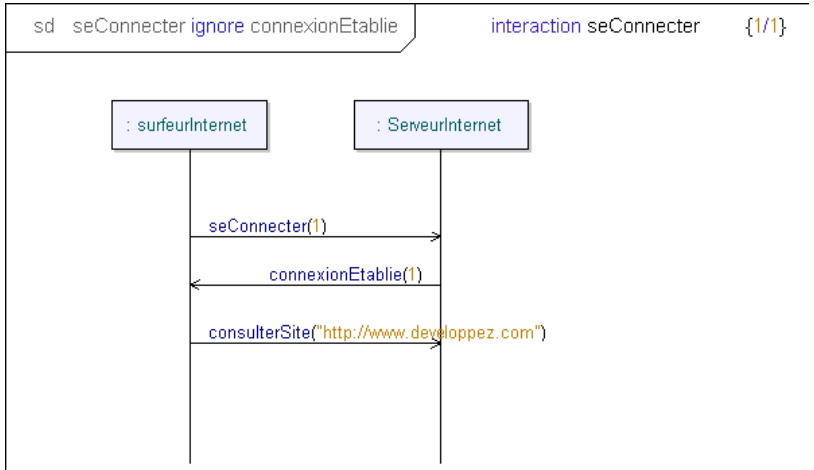


Exemple 4.11.

Ignore L'opérateur `ignore` (ignorer) indique qu'il existe des messages qui ne sont pas présents dans le fragment combiné. Ces messages sont en fait des messages que l'on peut qualifier d'insignifiants : intuitivement, ce sont des intéractions que l'on ne prend pas en compte. On peut aussi interpréter l'opérateur `ignore` désignant des intéractions pouvant intervenir à tout moment dans le flot des intéractions du diagramme de séquence.

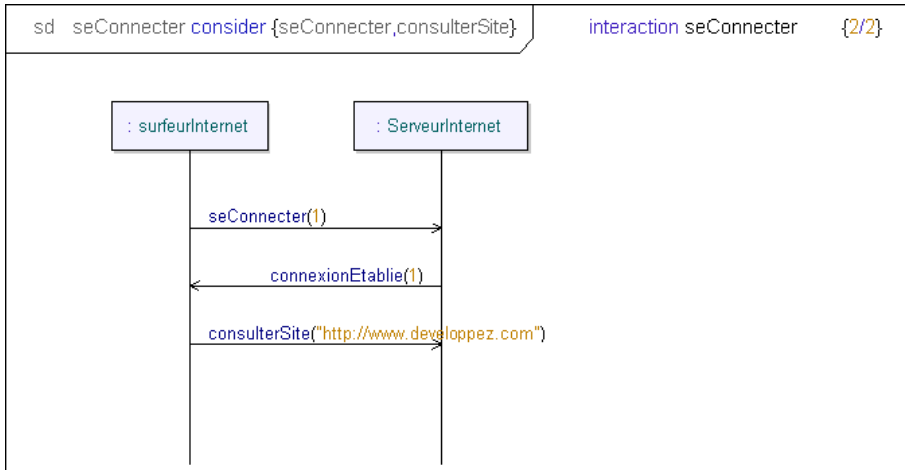
Exemple :Le message `connexionEtablie` est spécifié comme ignorée. On considère que la séquence est tout de même correcte si jamais lors de l'exécution ce message n'apparaissait pas.

Exemple 4.12.



Consider Au contraire, l'opérateur `consider` (considérer) désigne les interactions à prendre en compte dans la séquence. On peut imaginer que ce genre de construction soit plus particulièrement utilisé dans des profils orientés tests.

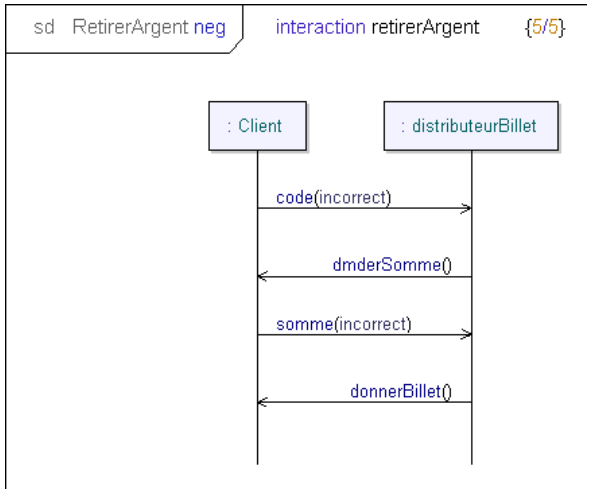
Exemple 4.13.



Neg L'opérateur `neg` (négatif) désigne un ensemble d'intéractions invalides.

Exemple : On ne souhaite pas que l'utilisateur puisse obtenir des billets avec un code erroné et une somme demandée incorrecte.

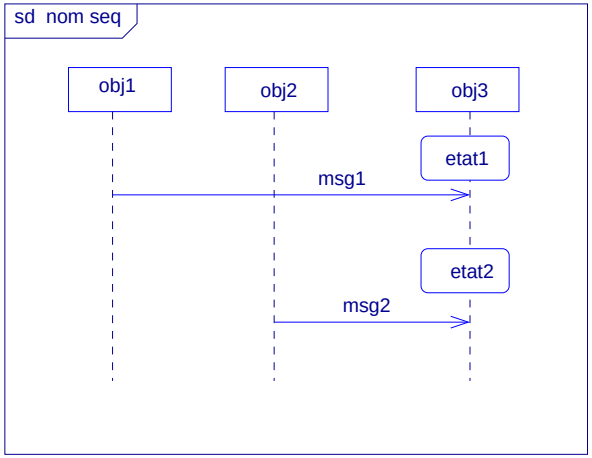
Exemple 4.14.



Invariant Il est possible d'indiquer les "lignes de vie" des entités des contraintes. Ceci est appelé "state invariant" dans la norme UML2.0. Attention, on parle d'état mais cela peut être tout simplement une valeur d'un attribut. Cette contrainte est considérée comme évaluée à l'exécution. Elle est évaluée immédiatement avant l'exécution de la prochaine intéraction de telle manière que toutes les actions qui ne sont pas explicitement modelisées soient considérées comme exécutées. Ces contraintes sont représentées par un état ou par un texte pouvant ressembler à

```
{ NomEntite.Attribut1==0}.
```

Exemple 4.15.



Exercice 4.3. *Syntaxe diag. de sequence*
On considère les interactions suivantes.

1. *Un objet* `o1` *de la classe* `CA` *crée un objet* `o2` *de la classe* `CB`*, puis lui envoie un message* `init()`.
2. *Selon la valeur de l'attribut* `x: integer` *de* `o1`*, les interactions suivantes ont lieu :*
   (a) *si* `x` *est strictement positif,* `o2` *envoie le message* `faitCeci()` *à un objet de la classe* `CC`*;*
   (b) *si* `x` *est nul,* `o2` *fait appel à la méthode* `faitCela()` *sur lui-même ;*

4.3 Diagramme de communication

Contrairement à un diagramme de séquence, un diagramme de communication rend compte de l'organisation spatiale des participants à l'interaction, il est souvent utilisé pour illustrer un cas d'utilisation ou pour décrire une opération.

4.3.1 Lignes de vie

Les lignes de vie sont représentées par des rectangles contenant une étiquette dont la syntaxe est :
`[<nom_du_role>] : [<Nom_du_type>]`

Au moins un des deux noms doit être spécifié dans l'étiquette, les deux points (:) sont, quand à eux, obligatoire.

4.3.2 Connecteurs

Les relations entre les lignes de vie sont appelées connecteurs et se représentent par un trait plein reliant deux lignes de vies et dont les extrémités peuvent être ornées de multiplicités.

4.3.3 Messages

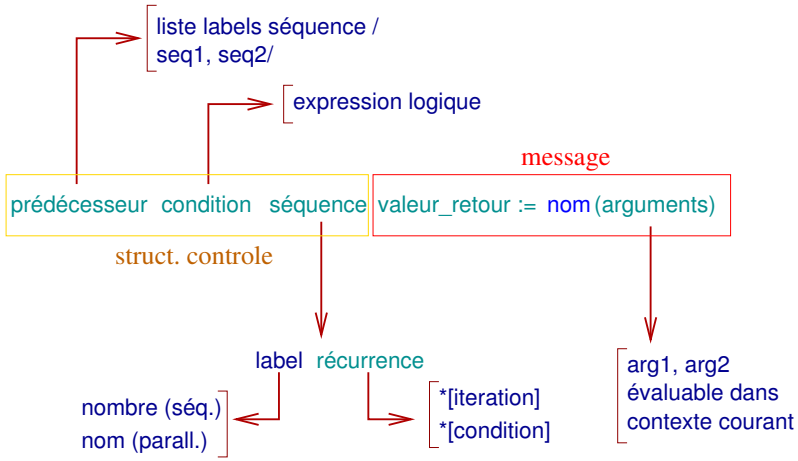


Figure 4.14.

Structures de contrôle : exemples

condition		récur. message
$[x > 0]$		: msg()
	$*$	: msg()
	$*$	$[x > 0]$: msg()
$[x > 0]$	$*$	$[i := 1..n]$: msg()

Exercice 4.4. *Tracer de graphes*

On considère un logiciel qui permet de tracer des graphes. Un objet g*, de la classe* Graph*, fait appel au service* draw(in c : Color) *sur un objet de la classe* Link *pour le dessiner en rouge.*
Représentez ceci sous forme d'un diagramme UML.

Exprimez également qu'un objet o1 *fait appel au service* calculer(in x : real, in i : integer) : real *d'un objet de la classe* C1*, si* x *est strictement positif, pour les valeurs de* $i, 0 \leq i \leq p$ *et que le résultat du calcul est stocké dans l'attribut* y *de* o1*.*

4.3.4 Exemples

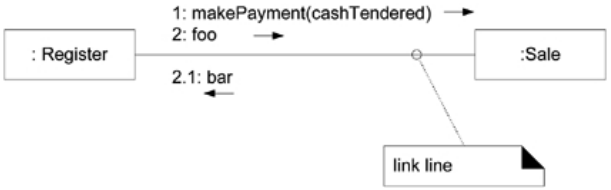


Figure 4.15. *Exemple*

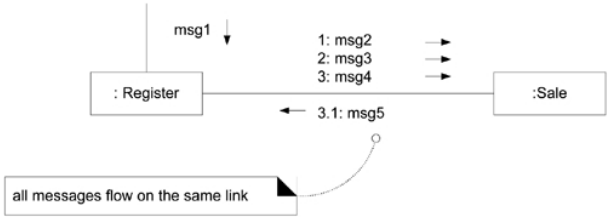


Figure 4.16. *Exemple*

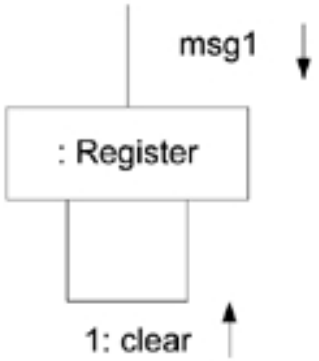


Figure 4.17. *Exemple*

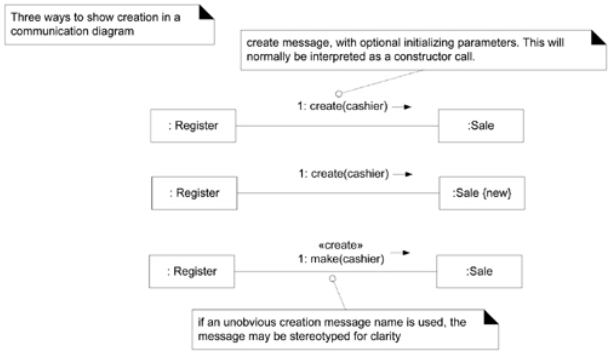


Figure 4.18. *Exemple*

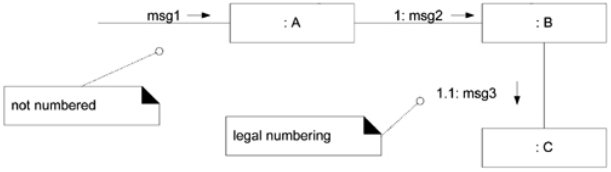


Figure 4.19. *Exemple*

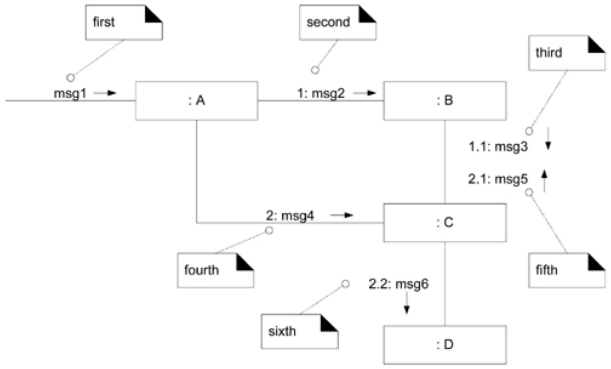


Figure 4.20. *Exemple*

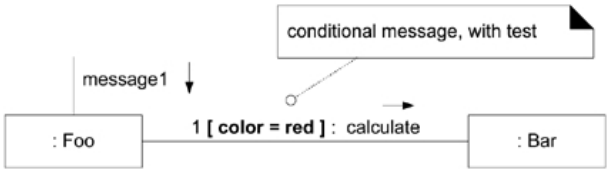


Figure 4.21. *Exemple*

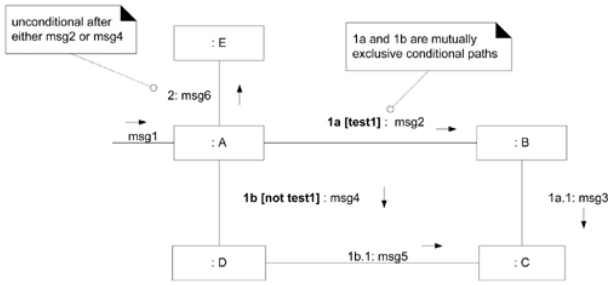


Figure 4.22. *Exemple*

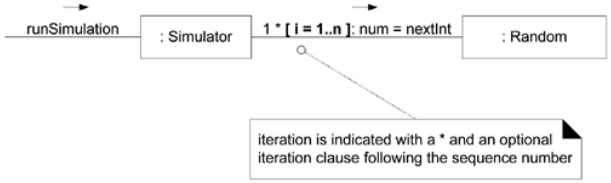


Figure 4.23. *Exemple*

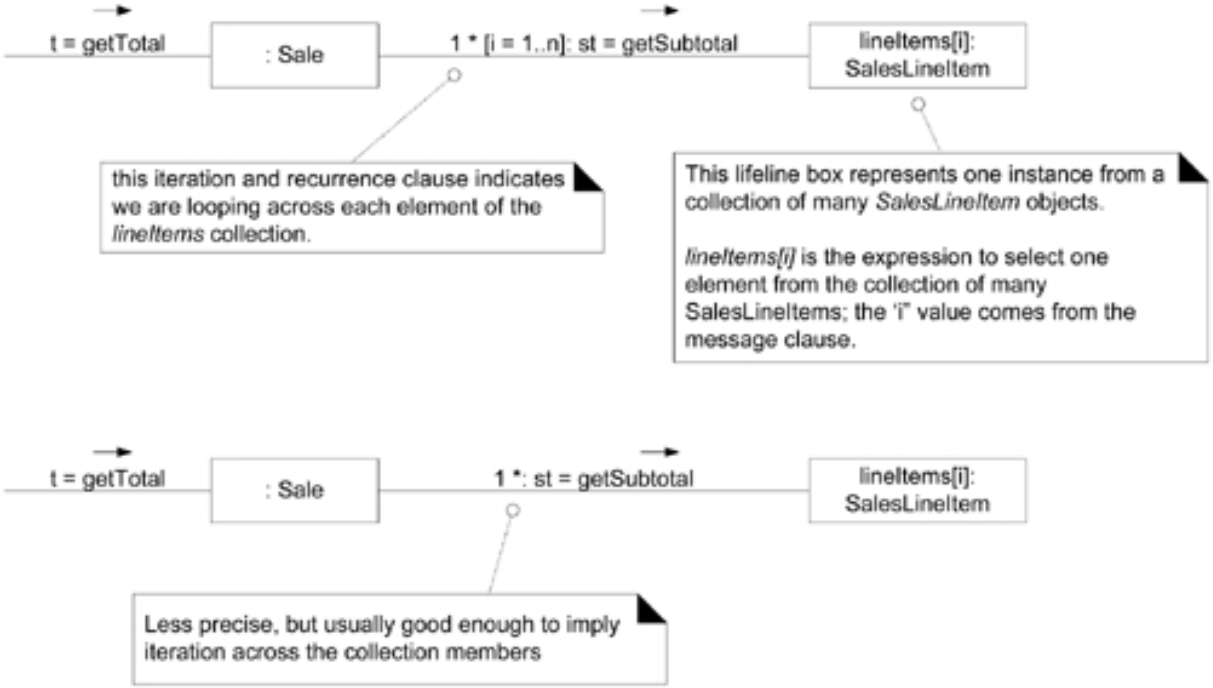


Figure 4.24. *Exemple*

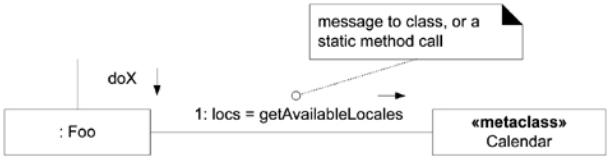
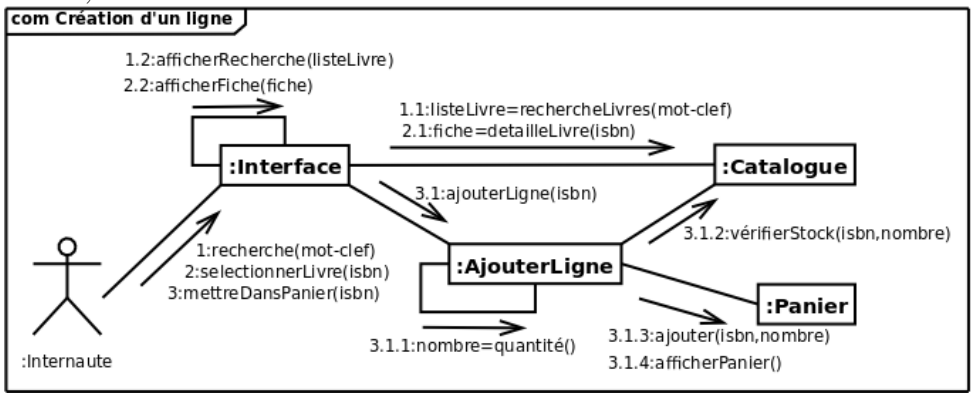


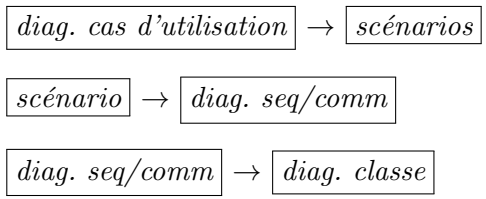
Figure 4.25. *Exemple*

Figure 4.26. *Diagramme de communication illustrant la recherche puis l'ajout, dans son panier virtuel, d'un livre lors d'une commande sur Internet*



4.4 Lien avec le diagramme de classes

Figure 4.27. *Du diag. de cas d'utilisation au diag. de classe*



4.5 Utilisation

Quand utiliser le modèle d'interaction ?

- ▷ Lorsque vous souhaitez visualiser le comportement de plusieurs objets dans un seul cas d'utilisation

Attention

- ▷ Si vous souhaitez visualisez le comportement d'un même objet dans plusieurs cas d'utilisation : utiliser le **diagramme d'états-transitions**
- ▷ Si vous souhaitez visualisez un comportement réparti sur plusieurs cas d'utilisation : utiliser le **diagramme d'activité**

4.6 QCM

4.6.1 Communication

1. Dans un modèle d'interaction un message correspond à : _____

   (a) une demande de service
   (b) un paramètre d'une opération
   (c) une structure de contrôle
   (d) une donnée en attente de traitement

2. Le fait que le message diviserPar(x) n'est émis que si x est non nul s'exprime par : .

   (a) *[x!=0]: diviserPar(x)
   (b) [x!=0]*: diviserPar(x)
   (c) diviserPar(x) {x!=0}
   (d) diviserPar(x!=0)
   (e) [x!=0]: diviserPar(x)

3. Dans la syntaxe d'un message, une clause de récurrence indique : _____

   (a) l'ordre dans lequel les messages sont émis
   (b) qu'un message est émis plusieurs fois
   (c) une communication asynchrone
   (d) que c'est un message de création
   (e) que le destinataire est récurrent

4. Un modèle de collaboration exprime tout sauf : _____

   (a) la réalisation d'un cas d'utilisation
   (b) des relations particulières entre objets
   (c) des objets en interaction
   (d) des échanges de messages entre objets
   (e) des relations particulières entre classes

4.6.2 Séquence

1. Dans un diagramme de séquence, un invariant d'état indique : _____
   (a) un état pris par un objet à un moment donné de l'interaction
   (b) un attribut en mode {readOnly}
   (c) une alternative dans l'interaction
   (d) une condition de réalisation de l'interaction
   (e) qu'un objet ne peut pas changer d'état au cours de l'interaction
2. Dans un diagramme de séquence, une ligne de vie représente : _____
   (a) rien du tout, c'est un piège
   (b) l'envoi de messages entre objets
   (c) l'exécution d'un comportement par un objet
   (d) l'existence d'un objet au cours de l'interaction
3. Dans un diagramme de séquence, un fragment d'interaction exprime : _____
   (a) une partie d'un objet
   (b) une structure de contrôle
   (c) un échange d'information entre deux objets
   (d) l'état d'un objet

Labo

5 Diagramme de classes (4 UC)

Sommaire

5.1 Classes

5.1.1 Carnet d'adresse

On s'intéresse à une version simple du référencement de personnes dans un carnet d'adresse à usage personnel. La figure ci-dessous est un exemple de fiche (tirée d'une application existante).

À partir de cet exemple de fiche, déduire les propriétés de la classe `Personne` dans ce système d'information.

5.1.2 Date

La classe `Date` a pour responsabilité de maintenir les informations relatives à une date : le jour, le mois et l'année.

La classe `Date` offre les services suivants.

1. `isBefore` évalue si la date est antérieure à une date donnée.
2. `isAfter` évalue si la date précéde à une date donnée.
3. `compareTo` permet de savoir si la date est avant, égale ou après une date donnée.

4. `parse` permet d'initialiser les champs d'une date à partir d'une chaîne de caractères de la forme DD-MM-YYYY.
5. `toString` est l'opération inverse.
6. `addDays` ajoute n jours à la date.

Représentez les différentes propriétés de cette classe `Date` en UML.

5.2 Associations

5.2.1 Mail

Pour être acheminé et exploité correctement, un message (= un *mail*) doit possèder un certain nombre de propriétés qu'il s'agit ici de modéliser.

1. Un message est nécessairement créé par un utilisateur (on simplifie) qui est représenté sous la forme d'une `identité`. Cette identité fait référence à un compte de messagerie.
2. Pour chaque identité, l'utilisateur peut définir une ou plusieurs signatures pour ses messages, dont l'une sera ajoutée au corps du message lors de la rédaction de celui-ci.
3. Un message peut avoir plusieurs destinataires (identités) ; il est également possible de préciser une liste d'identités auxquelles le message est envoyé en copie.
4. Un message possède un champs `objet` et un `corps` ; il peut aussi contenir des documents attachés (fichiers).

Représentez toutes ces informations sous forme d'un modèle de classes UML.

5.2.2 Vols

L'étude porte sur un système simplifié de réservation de vols pour une agence de voyage. Les entretiens avec les experts du métier ont permis de recueillir les connaissances suivantes sur le domaine de la réservation. Elle consituent le point de départ de l'analyse des exigences fonctionnelles du système.

1. Les compagnies aériennes proposent différents vols.
2. Un vol est ouvert à la réservation et refermé sur ordre de la compagnie.

3. Une réservation concerne un seul vol et un seul passager.
4. Une réservation peut être annulée ou confirmée.
5. Une réservation est facturée au client à un certain prix, dont le calcul est assez compliqué.
6. Un vol a un aéroport de départ et un aéroport d'arrivée.
7. Un aéroport est identifié par un code (p. ex. BES, CDG, ORY) et par la ville à laquelle il est rattaché.
8. Un vol a un jour et une heure de départ, et un jour et une heure d'arrivée.

Représentez ces connaissances sous forme d'un modèle de classes UML.

Par définition une agence de voyage gère des... voyages.

1. Un voyage peut être composé d'une suite de vols, par exemple Brest $\rightarrow$ Paris, Paris $\rightarrow$ Tokyo et Tokyo $\rightarrow$ Sydney.
2. Il est possible de calculer la durée totale d'un voyage, ainsi que son prix.
3. L'agence édite un dossier de voyage qu'elle remet au client. Ce dossier est identifié par un code, le nom du correspondant dans l'agence et récapitule toutes les informations sur les vols.

Complétez votre modèle de classes UML avec ces éléments.

Comme cela a déjà été mentionné, la gestion tarifaire est compliquée. Il y a un tarif de base pour chaque trajet, des taxes d'aéroport et des majorations qui dépendent des éléments suivants :

1. la classe de prestation ;
2. le poids maximum autorisé pour les bagages ;
3. la période.

Complétez votre modèle de classes UML avec ces éléments.

5.3 Relations

5.3.1 Module pédagogique

On considère ici une partie d'un système d'information concernant les modules d'enseignement constituant un programme pédagogique. On suppose que tout l'enseignement se fait sous forme de modules et qu'il s'inscrit dans le cadre de l'ECTS.

1. Un module est identifié par un code alphanumérique et un nom.
2. Chaque module donne droit à des crédits ; ce nombre est non modifiable (3 par défaut).
3. Les modules ne sont pas tous indépendants : un module peut nécessiter des pré-requis (d'autres modules). Par conséquent, un module donne la possibilité d'en suivre d'autres.
4. Les modules sont organisés en *unités d'enseignement* qui sont de 2 types : les cours magistraux et les travaux dirigés. Chaque unité d'enseignement est caractérisée par un nom et une durée (en heures). Les cours magistraux et les travaux dirigés ont des taux d'encadrement différents, respectivement 1.5 et 1.0.
5. Chaque unité d'enseignement est assurée sous la responsabilité d'un enseignant. Il y a aussi un enseignant responsable par module, nécessairement un des enseignants intervenant dans le module.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

5.3.2 Gestion de projet

On s'intéresse à la modélisation des informations nécessaires à la gestion de projet : planification des actions, calcul de l'avancement et des coûts, affectation du personnel.

A chaque projet est alloué un budget et une date de début ; des personnes sont affectées au projet et un chef de projet est désigné (parmi ces personnes). Il est possible de calculer la rentabilité du projet (différence budget - coûts) et son avancement à une date donnée (pourcentage) ; la manière de faire ces calculs ne fait pas partie de cette étude.

La planification d'un projet consiste à identifier toutes les actions qui devront être réalisées et leurs enchaînements ; ainsi, on peut connaître toutes les actions devant être réalisées avant une action donnée et, symétriquement, toutes celles qui peuvent être réalisées après. Les actions sont de deux

types : soit il s'agit de tâches à réaliser ; soit il s'agit de jalons. Une tâche a une durée alors qu'un jalon est réalisé est une action à réaliser à une date donnée.

Le calcul du coût d'une tâche nécessite de connaître le temps passé à la réalisation de cette tâche par personne. à chaque personne est associée un niveau de qualification, chaque qualification correspondant à un coût par unité de temps (coût unitaire).

Lorsqu'une personne réalise une tâche, on enregistre le temps associé.

Modélisez les éléments de ce système d'information sous la forme d'un diagramme de classes UML : identifiez les classes, leurs relations et les propriétés de ces éléments. .

5.3.3 Publications

On s'intéresse à un système d'information pour le référencement de publications scientifiques. Le logiciel permet à un scientifique de référencer toutes les publications qu'il utilise dans ses travaux de recherche.

Les scientifiques peuvent publier les résultats de leurs travaux dans différents types de publication. Chaque publication a un titre. Elle est écrite par une série d'auteurs (l'ordre est significatif). On indique aussi son année de parution. La recherche de publications peut se faire par des mots-clés.

Représentez le lien entre publications et auteurs.

Quand on écrit une publication scientifique, on fait toujours référence à des travaux existants en citant les publications dans lesquelles ces travaux ont été publiés. Le logiciel permet de référencer les publications et de savoir quelles sont les publications qui sont citées dans une publication et, symétriquement, dans quelles publications une publication est citée.

Représentez ce système de référencement entre publications. Est-il possible de connaître le nombre de publications citées dans une publication ? Est-il possible de connaître combien de fois une publication a été citée ? Est-il possible de connaître les publications dans lesquelles tel auteur est cité ? Est-il possible de connaître le nombre de ces publications ?

Le logiciel permet de connaître toutes les publications écrites par un auteur. On peut aussi lister l'ensemble des publications d'un laboratoire de recherche.

Modifiez votre modèle pour ajouter ces propriétés. Vérifiez que l'on peut calculer le nombre moyen de publications par auteur pour le laboratoire.

Les publications référencées sont de différentes natures : articles dans des journaux scientifiques, publications dans des actes de conférence, thèses, rapports de recherche.

Dans le cas des articles dans des journaux scientifiques, en plus des informations déjà mentionnées, on indique le nom du journal, le numéro du volume, le numéro du journal et les pages de l'article (première et dernière).

Pour les articles dans les actes de conférence, on précise le nom de la conférence, la ville dans laquelle elle s'est tenue, ainsi que les pages de l'article.

Représentez ces différents types de publication et leurs propriétés.

Dans le cas d'une thèse, on précise le nom de l'université qui l'a délivré, la spécialité, la date de soutenance et la composition du jury : président (un seul) , rapporteurs (au moins 2), examinateurs (entre 2 et 4), en indiquant leurs grades (docteur, professeur). On enregistre aussi le nom du directeur de thèse qui ne fait pas nécessairement partie du jury.

Le président du jury ne peut pas être un des rapporteurs : il s'agit nécessairement d'un des examinateurs et il ne peut pas s'agir du directeur de thèse.

Représentez ces diférentes propriétés sur les thèses dans votre modèle de classes. Est-il possible de connaître le nombre de thèses dirigées par un membre d'un laboratoire donné ? Est-il possible de connaître les publications citées dans une thèse ?

5.4 Interface

5.4.1 Collections d'objet

Pour la majorité des langages de programmation, il existe une bibliothèque de classes qui assurent la gestion de collections d'objets. Il existe différents types de collection et pour chacun diverses

possibilités de réaliser les services de ces collections.

Ces bibliothèques sont généralement conçues en distinguant des interfaces qui définissent les services possibles sur une collection et des classes qui supportent la réalisation de ces services. On s'intéresse ici à un cas fictif inspiré d'un cas réel.

1. L'interface `Collection` déclare l'ensemble des services réalisables sur une collection d'objets, tels que l'ajout d'un objet, le décompte des objets, le test de l'existence d'un objet dans la collection.
2. Pour le parcours des éléments de la collection l'interface `Collection` utilise les services de l'interface `Iterator`.
3. L'interface `Set` spécialise l'interface `Collection`.
4. Les deux classes `HashSet` et `TreeSet` réalisent l'interface `Set`.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

1. Les autres types de collections, telles que les listes, sont conçues de la même façon, avec deux implémentations, `ArrayList` et `LinkedList`.
2. `LinkedList` offre des services particuliers tels que l'ajout d'un élement en tête de liste, l'ajout d'un élement en fin de liste, la récuperation du premier élement, du dernier élement.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

On considère la modélisation d'une famille d'algorithmes de tri : tri par insertion (*insertion sort*), tri par arble binaire (*BinaryTreeSort*...). Tous ces algorithmes fonctionnent sur un ensemble d'objets du même type. Avant réalisation de l'opération de tri, l'ensemble est dans un ordre quelconque, après, les élements sont ordonnés. Le tri peut être fait par ordre croissant ou décroissant. Pour pouvoir remplir le service qui est sous sa responsabilité, l'algorithme de tri doit manipuler des objets qui sont comparables. Ces derniers doivent donc offrir les services de comparaison : `lesserThan`, `gretterThan`, `equals`. Une fois conçus et réalisés, les algorithmes de tri doivent pouvoir travailler sur des objets de type quelconque, pourvu qu'ils soient comparables : on peut imaginer trier entre eux des objets tels que des personnes, des dates, des rectangles...

Proposez un modèle de classes UML qui répond à ces différentes exigences

5.5 Synthése

5.5.1 AReViLib3d

La conception d'une application graphique repose sur l'utilisation d'une bibliothèque de classes pour le rendu graphique 3D qui s'appelle ARéVi.

Dans cette bibliothèque, les objets graphiques sont de la classe `Object3D`. Les objets sont localisés dans l'espace, propriété qui est définie dans la classe `Base3D` dont hérite `Object3D`. La classe `Base3D` offre les services publics suivants :

1. `locate` : positionnement en un point donné spécifié par un ensemble de 3 coordonnées (nombres réels) ;
2. `move` : déplacement selon un vecteur spécifiée sous la forme d'une `Base3D` ;
3. `location` : obtention de coordonnées de l'objet, sous forme de 3 nombres réels ;
4. `attachTo` : attachement à une autre `Base3D` (en conséquence les 2 `Base3D` sont liées, le déplacement de l'une se répercute sur l'autre).

La classe `Object3D` offre des services qui permettent de définir comment un objet 3D réagit quand on interagit avec lui dont :

1. `onMouseInteraction` : définition du comportement quand on interagit avec l'objet à l'aide d'une souris. Cette opération utilise les services d'un `Interactor` qui est ici une interface. On lui fournit aussi le numéro du bouton actionné et on indique si le bouton est pressé (oui ou non).

L'interface `Interactor` définit le service `getEvent` qui fournit en sortie un objet de la classe abstraite `Event`. La classe concrète `BasicMouseInteractor` réalise cette interface.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

La conception porte sur un objet complexe qui permet de représenter graphiquement des surfaces paramétrées. Ceci est supporté par un ensemble de classes regroupées dans le package `grapher` qui utilise les classes de la bibliothèque ARéVi (voir exercice correspondant).

La structure d'un grapher (classe `Grapher`) est la suivante :

1. il est composé de 3 axes, `Ox`, `Oy` et `Oz` ;
2. à chaque axe on associe un domaine de valeur qui est défini par un min, un max (nombres réels) et un nombre de valeurs ;

3. le grapher peut afficher une ou plusieurs surfaces (chaque surface référence le grapher dans lequel elle est représentée) ;
4. 2 domaines sont associés à chaque surface : x et y (ceci permet de savoir qu'elle partie de la surface est visible).

Les axes et les surfaces sont des `Object3D` particuliers, le grapher une `Base3D`.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

5.5.2 Mail 2

La gestion locale des messages par l'utilisateur repose sur la notion de compte de messagerie (souvent appelé « compte mail ») et de boîtes de messages.

Un compte de messagerie offre différents services.

1. Il permet d'authentifier l'utilisateur ; les informations nécessaires sont un nom et un mot de passe.
2. On peut relever le courrier d'un compte : chargement des messages depuis le serveur de messagerie vers le client.
3. On peut aussi envoyer un message : transfert du message à tous ses destinataires.

Pour effectuer la réception et l'émission des messages, le compte de messagerie s'appuie sur les services de 2 classes abstraites : `ServeurReception` et `ServeurEmission`.

1. Pour le `ServeurReception`, on précise la fréquence de relevé des messages (60 secondes par défaut) et si les messages relevés sont détruits ou non sur le serveur distant. Il possède une opération abstraite `chargerMessages` qui crée localement les messages récupérés.
2. Côté émission, on n'envisage ici qu'un type de serveur : `ServeurSMTP`. Celui-ci assure la transmission d'un message à un destinataire et fournit une indication permettant de savoir si l'envoi s'est bien passé ou non.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

Localement les messages sont stockés dans des boîtes de messages. Il existe plusieurs types de boîtes de messages qui se différencient par leurs comportements. Il s'agit donc de bien identifier les services correspondants.

1. Pour chaque compte, il y a une boîte de réception, une boite d'envoi, une corbeille.
2. L'utilisateur peut ausi créer autant de boîtes de messages personnelles qu'il le désire. On les désigne par le terme *boîte locale*.
3. Quand l'utilisateur rédige un nouveau message, il est placé dans la boîte d'envoi. Quand un message est reçu, il est stocké dans la boîte de réception.
4. On peut supprimer un message d'une boîte. Si cette boîte n'est pas la corbeille, le message est placé dans la corbeille. La corbeille peut être vider en une seule opération.
5. On peut déplacer un message entre 2 boîtes locales, entre la boîte d'envoi et une boîte locale et entre la boîte de réception et une boîte locale.

Représentez toutes ces informations sous forme d'un modèle de classes UML.

5.5.3 Voisinage Cellule

Dans de nombreuses simulations, telles que les jeux vidéo, l'espace est découpé en *cellules*. Les comportements des personnages nécessitent de connaître les cellules vers lesquelles ils peuvent se déplacer ou celles dont ils peuvent percevoir le contenu. Ceci repose sur la notion de voisinage. Deux types de voisinage sont classiquement utilisés : le voisinage de MOORE (défini par les 4 cellules situées au nord, au sud, à l'ouest et à l'est) et le voisinage de VON NEUMANN (les 8 cellules adjacentes).

1. La classe `Cellule` implémente une opération de calcul des voisins qui retourne les cellules réellement atteignables. Pour cela, elle fait appel à un voisinage qui a pour responsabilité de retourner les cellules « voisines ». La cellule n'a pas à connaître le type d'objet qui rend ce service ; la responsabilité de la cellule est de déterminer les cellules qui sont effectivement accessibles dans l'ensemble des cellules qui constituent son voisinage (il peut y avoir des cellules « interdites »).
2. La notion de voisinage se définit par le service `calculerVoisins` qui prend en entrée un nombre entier qui indique la « profondeur » du voisinage (les voisins directs ou les voisins des voisins etc) et qui retourne les cellules faisant parties du voisinage.
3. Deux façons de rendre ce service sont définies : le voisinage de MOORE (`VoisinageMoore`), le voisinage de VON NEUMANN (`VoisinageVN`).
4. Les classes dans lesquelles les algorithmes de calcul de voisinage sont implémentés ne sont modélisées dans cette étude.

Q.5.1 Représentez toutes ces informations sous forme d'un modèle de classes UML.

5.5.4 Album Photos

On s'intéresse ici aux fonctionnalités d'un logiciel, appelé AlbumPhoto, qui permet de gérer des photos numériques, de les importer depuis un appareil photo numérique, de les organiser, de leur appliquer des retouches...

Le logiciel gère deux types d'item : les albums et les photos. Un album peut contenir d'autres albums ou des photos. Ces deux types d'item sont identifiés par un nom. Ils ont aussi comme propriétés leur date de création et la date de leur dernière modification (on suppose que l'on dispose d'une classe `Date`). Il est possible de calculer la taille d'un item (l'information est fournie sous la forme d'un entier, le nombre d'octets). Dans le cas d'une photo, il s'agit de la taille de son fichier et dans le cas d'un album, de la somme des tailles des items qu'il contient.

Modélisez cette structuration des données manipulées par le logiciel ainsi que leurs propriétés sous forme d'un diagramme de classes UML.

Le logiciel permet d'appliquer différents types de retouche aux photos. On peut appliquer plusieurs retouches à une même photo. À tout moment, on peut savoir qu'elles retouches ont été appliquées à une photo et dans quel ordre.

Comme bon nombre de commande dans ce type de logiciel, l'effet obtenu suite à l'application d'une commande n'est pas toujours celui qu'on espérait ! Pour cela, il est possible d'annuler une commande et de l'appliquer à nouveau. Les retouches implémentent ce mécanisme : elles peuvent être appliquées et annulées, services que peuvent offrir d'autres classes de l'application.

Un type particulier de retouche est l'application d'un filtre de couleur. Il consiste à changer les composantes R,G,B des pixels de la photo. Une autre retouche possible est de rogner la photo. Pour cela l'utilisateur définit un cadre (classe `Frame`) dont les propriétés sont les deux offsets en `x` et `y` des coins inférieurs gauche et supérieur droit. Appliquer cette retouche nécessite donc d'associer le cadre à la photo.

Modélisez ces propriétés qui supportent les fonctionnaliés du logiciel sous forme d'un diagramme de classes UML.

6 Diagramme de cas d'utilisation (2 UC)

Sommaire

6.1 EDT

On considère un logiciel de gestion des emplois du temps, nommé EDT, qui a, entre autres, comme fonction la planification des périodes d'enseignement. Cette plannification est faite par le directeur des études. Les étudiants et les professeurs peuvent consulter les emploi du temps. Tous les utilisateurs doivent se connecter avant d'accéder à l'application.

Représentez ces informations sous forme d'un diagramme de cas d'utilisation UML. Représentez les scénarios associés.

6.2 Station service

On s'intéresse à la modélisation des cas d'utilisation d'un logiciel de gestion d'une station service.

1. Un client de la station peut activer la distribution de carburant.
2. Il peut éventuellement activer une distribution « limitée » (il fixe alors un montant ou un volume de carburant)
3. Tout distribution de carburant conduit au paiement de la quantité livrée.
4. Le paiement constitue une sorte de "transaction commerciale" ; elle se fait sous le contrôle d'un système monétique.
5. Le paiement se fait par le client sous le contrôle du pompiste.

Exprimez toutes ces fonctionnalités sous forme d'un diagramme UML des cas d'utilisation.

6.3 Client mail

L'application client mail est un logiciel qui permet à un utilisateur de recevoir des messages (mails) de les gérer et d'en envoyer. Les principales fonctionnalités de cette application sont les suivantes :

1. L'utilisateur peut relever les messages d'un de ses comptes de messagerie ; pour cela, l'application doit être connectée au serveur de réception.
2. L'application permet à l'utilisateur de gérer localement ses messages : lecture des messages, création de boîtes de messages, rangement des messages, suppression de messages, définition de sa signature...
3. L'application permet d'envoyer des messages, ce qui impose d'être connecté à un serveur d'envoi de message.
4. Relever son courrier, de même qu'en envoyer, nécessite que l'utilisateur soit authentifié.

Exprimez toutes ces exigences fonctionnelles sous forme d'un diagramme UML des cas d'utilisation.

6.4 Copieur multi-fonctions

On s'intéresse ici à la modélisation des fonctionnalités d'un copieur multi-fonctions. Il s'agit d'un cas d'école qui ne correspond pas nécessairement à un système réel. Ce copieur multi-fonctions est un système 3 en 1 : photocopieur, imprimante, scanner.

1. Le copieur multi-fonctions permet de copier des originaux sur support papier.
2. S'il le souhaite, l'utilisateur peut activer un mode particulier permettant de ne copier qu'une zone de la page.
3. Deux fonctions d'impression sont disponibles : impression d'un document paginé sur support papier et impression sur disque (CD ou DVD) ;
4. Enfin , le copieur multi-fonctions permet de numériser un document.

5. La fonction d'impression, comme celle de numérisation, requiert le transfert des données avec un ordinateur.

Exprimez toutes ces exigences fonctionnelles sous forme d'un diagramme UML des cas d'utilisation.

6.5 Caisse enregistreuse

Le déroulement normal d'une caisse enregistreuse est le suivant :

- ▷ Un client arrive à la caisse avec des articles à payer.
- ▷ Le caissier enregistre le numéro d'identification de chaque article, ainsi que la quantité si elle est supérieure à 1.
- ▷ La caisse affiche le prix de chaque article et son libellé.
- ▷ Lorsque tous les achats sont enregistrés, le caissier signale la fin de la vente.
- ▷ La caisse affiche le total des achats.
- ▷ Le client choisit son mode de paiement.
    1. Liquide : le caissier encaisse l'argent reçu, la caisse indique la monnaie à rendre au client.
    2. Chèque : le caissier vérifie la solvabilité du client en transmettant une requêteà un centre d'autorisation via la caisse.
    3. Carte de crédit : un terminal bancaire fait partie de la caisse. Il transmet une demande d'autorisation en fonction du type de la carte.
- ▷ La caisse enregistre la vente et imprime un ticket.
- ▷ Le caissier donne le ticket de caisse au client.
- ▷ Lorsque le paiement est terminé, la caisse transmet les informations sur le nombre d'articles vendus au système de gestion des stocks.
- ▷ Tous les matins, le responsable du magasin initialise les caisses pour la journée.

1. Donnez un *diagramme de cas d'utilisation* de la caisse enregistreuse.
   (a) Identifiez les acteurs
   (b) Identifiez les ≪ gros ≫ cas d'utilisation
   (c) Réalizer le diagramme complet
2. Traitez le cas d'un passage en caisse en ne considérant que le paiement en liquide (identification, *scénarios* et enchaînements alternatifs).

6.6 Réservations de spectacles

L'étude porte sur un logiciel de gestion de spectacles (appelé ReSpect) pour un centre culturel. L'administrateur du centre a la responsabilité du choix des spectacles et de leur mise en vente, qui se fait uniquement par réservation. Le logiciel ReSpect assure la gestion des réservations et la mise en vente des places.

Avant de proposer un spectacle à la vente, le centre culturel doit d'abord sélectionner une œuvre et en acquérir les droits de représentation puis engager des artistes. Il faut ensuite construire les décors, faire les costumes et organiser des répétitions. Si c'est un succès, il faut alors faire la publicité du spectacle, programmer les différentes représentations (qui ont lieu dans une des salles gérées par le centre culturel à des dates et heures à déterminer) et le mettre à la vente. Parallèlemment, des répétitions doivent être organisées afin de rôder le spectacle.

Un spectacle peut donner lieu à plusieurs représentations ; dans les salles, les places sont numérotées. Différents modes de réservation sont possibles :

- ▷ réservation individuelle pour un spectacle ;
- ▷ réservation pour un groupe de personnes pour un même spectacle ;
- ▷ souscription individuelle à une série de spectacles.

La réservation se fait soit via un terminal de télé-paiement, appelé kiosque – qui est un dispositif externe – ou en passant par un opérateur qui saisit les choix du client. Dans tous les cas, le paiement se fait par carte bancaire. Lorsque la réservation est effectuée par le kiosque [2], le client doit au préalable introduire sa carte bancaire dans le terminal ; ce dernier permet d'éditer le ticket de carte. Dans le cas d'une réservation individuelle, le client peut indiquer une liste de choix de dates. Si la réservation est faite par l'opérateur, celui-ci saisit les informations que le client lui transmet (voir ci-dessous)

2. Le kiosque constitue une interface entre le client et le serveur de réservation ReSpect.

son numéro de carte bancaire et sa date d'expiration. Les informations nécessaires à une réservation sont le nom et le numéro de téléphone du client puis les caractéristiques du spectacle (nom, date, heure). Tous les paiements font l'objet d'un contrôle auprès d'un service bancaire auquel est connecté ReSpect. Les clients peuvent éventuellement changer la date de la représentation qu'ils ont choisie (uniquement dans le cas d'une réservation individuelle). Cette opération n'est plus possible à moins de deux semaines de la date du spectacle. ReSpect permet à l'administrateur du centre culturel de gérer la programmation des spectacles (mise en vente, annulation de représentation, prolongation) et de suivre le taux de réservation des représentations.

L'objectif ici est d'obtenir une vue générale des cas d'utilisation et de préciser certains scénarios de réalisation. Pour cela, vous devez analyser la description de l'activité de programmation des spectacles et de gestion des réservations telle qu'énoncée ci-dessus et dégager les cas d'utilisation du logiciel et les acteurs. Pour cela, il faut délimiter les fonctionnalités supportées par le logiciel. Voici quelques indications pour commencer :

1. toute réalisation du cas d'utilisation "réservation" implique celle du cas "paiement" ;
2. le cas d'utilisation "paiement" est une spécialisation du cas "transaction" ;
3. dans le cas particulier d'une réservation pour une série de spectacles, il faut réaliser le cas d'utilisation "souscription abonnement" ;
4. la réservation est faite par un opérateur, une opératrice.

Établissez le diagramme des cas d'utilisation du logiciel ReSpect.

Une fois les cas d'utilisation et les acteurs identifiés, l'étape courrante est de donner une description sommaire de chacuns : objectifs de la réalisation des cas, identification des scénarios, rôles des acteurs.

Donnez une description sommaire scénarios associés cas d'utilisation du logiciel ReSpect.

7 Diagramme d'interactions (4 UC)

Sommaire

7.1 Photo import

On s'intéresse ici aux interactions entre le module d'import de photos du logiciel AlbumPhoto et un appareil photo numérique lors de l'import des photos, une des fonctionnalités majeure de ce type de logiciel. Un appareil photo numérique est représenté ici par un classifier nommé `Camera`.

Question 7.1 : modélisez l'existence de cette fonctionnalité dans le logiciel AlbumPhoto ; faites figurer sur votre diagramme que sa réalisation est sous le contrôle de l'utilisateur et requiert l'existence d'une caméra connectée au système.

Les interactions entre le module d'import, nommé `PhotoLoader` et la `Camera` se déroulent de la manière suivante.

1. La camera envoie de manière asynchrone un message `init` indiquant le nombre de photos transférables.
2. L'utilisateur confirme l'import en envoyant le message `startImport` qui porte le nom du nouvel album photo qui correspond à l'import.
3. Le module d'import crée l'album correspondant et effectue l'import, photo par photo, comme suit.

(a) La camera envoie d'abord l'identifiant de la photo. Si la photo existe déjà, elle n'est pas traitée (testé par l'opération `exist(in id: PhotoId): boolean`).

(b) Le fichier de chaque photo est ensuite récupéré par le module d'import en appelant l'opération `retrievePhoto(in id: PhotoId): File` de `Camera`. Il y a création d'un objet `Photo` pour chaque photo traitée.

(c) L'opération de création de la photo enregistre celle-ci (opération `save`) sur l'ordinateur sur lequel s'exécute `AlbumPhoto`.

(d) Le tranfert est interrompu à tout moment si la caméra envoie le message `TransfertError` au module d'import.

Question 7.2 : modélisez la réalisation de la fonction d'import sous la forme d'un diagramme de séquence UML.

7.2 Polygone

On se place dans le contexte de la conception d'un logiciel permettant de dessiner des formes géométriques et plus particulièrement au tracé de polylines, c'est à dire des formes composées de segments de droite reliant des points.

Définition d'un segment de droite. Un segment de droite (classe `Segment`) est l'élément de base d'une polyligne (classe `Polyline`). Il est composé d'un trait (classe `Line`) reliant deux points (classe `Point`), le trait référence aussi ces deux points.

Représentez, sous forme d'un diagramme d'objet d'UML, une instance de la classe `Segment` en faisant apparaître les relations entre les instances avec lesquelles cette instance est liée.

Création d'un segment de droite. La classe `Segment` dispose d'une opération de création (`create`) ayant comme paramètres deux paires de valeurs entières qui correspondent chacune aux coordonnées des points formant les extrémités du segment à créer. La création d'un segment a pour effet de créer ces deux points et le trait qui les relie.

Reprenez le diagramme précédent en le transformant en diagramme de communication afin de représenter ce processus de construction.

Création d'une polyligne. L'éditeur graphique permet de dessiner une polyligne en ajoutant progressivement les points qui forment chacun de ses sommets. L'opération `newEdge` permet ainsi d'ajouter un sommet (*edge*) à une polyligne. S'il ne s'agit pas du premier point de la polyligne, cette opération a pour conséquence la création d'un segment reliant le dernier point entré et le nouveau. La classe `Polyline` dispose, entre autre, des opérations `addEdge`, `lastEdge`, `nbEdge`.

Créez un diagramme de communication afin de représenter la réalisation du service `addEdge` par une instance de `Polyline`.

Calcul de la longueur d'une polyligne. Cette longeur est définie comme la somme des longueurs des segments qui composent une polyligne ; elle est nulle dans le cas où la polyligne contient moins de deux points.

Créez un diagramme de communication afin de représenter la réalisation du calcul de la longeur d'une polyligne (opération `length`).

Diagramme de classe. L'étude que vous venez de faire sur les opérations relatives à la construction et à la manipulation des polylignes permet de dégager bon nombre des propriétés des classes relatives à ce sous-système.

Construisez le diagramme de classes correspondant, en faisant apparaître toutes les propriétés et relations des classes que vous avez identifiées.

7.3 Appel d'offre

Il existe un type de système informatique que l'on appelle les systèmes multi-agents. On utilise ce type d'architecture quand différents composants d'un système sont distribués sur un réseau informatique et qu'on cherche une forte robustesse du système. Les composants, que l'on nomme *agents*, effectuent des tâches pour leur propre compte et quand l'un d'entre eux n'a pas les ressources

nécessaires pour exécuter une des tâches qui lui incombe, il fait appel aux services d'autres agents selon un protocole bien défini.

Le protocole le plus utilisé dans les systèmes multi-agents est inspiré de celui des appels d'offre que se font les entreprises : lorsqu'une personne (un agent) dans une entreprise veut sous-traiter une tâche, il soumet cette demande à un gestionnaire de l'entreprise qui envoie un appel d'offre à des prestataires de service. Si ces derniers sont intéressés, ils renvoient une offre de service au demandeur en précisant leurs conditions (le prix entre autre). Le gestionnaire de l'entreprise dépouille ensuite les offres et, si une d'elle lui convient, il passe un contrat avec prestataire correspondant. En réalité c'est un peu plus compliqué que cela, mais c'est cette idée qui est retenue dans les systèmes multi-agents (avec quelques variantes).

L'objectif de cette étude est de modéliser une version légèrement simplifiée de ce protocole.

Emission d'un appel d'offre (AO)

Cette première partie du protocole commence par l'expression d'une demande et se termine par la soumission de l'offre à des prestataires potentiels.

1. Tout débute par un agent qui veut faire réaliser une tâche. Cet agent – le demandeur – crée une demande de service (`DdeService`) en précisant le type de tâche et le délai (un entier).
2. Il soumet ensuite cette demande à un `Gestionnaire` (une sorte d'agent). Ce gestionnaire créé un appel d'offre à partir de la demande en ajoutant des exigences (ici un prix).
3. Le gestionnaire dispose d'un annuaire de prestataires ; il peut lancer une requête sur cet annuaire pour obtenir l'ensemble des prestataires capables de réaliser la tâche demandée.
4. Le gestionnaire soumet l'appel d'offre à chaque prestataire sélectionné. Les `Prestataires` sont des agents particuliers.

Représentez cette première étape du protocole sous forme d'un diagramme de séquence d'UML.

Réponse à un AO : formulation d'une offre de service (OS)

On modélise ici le traitement d'un appel d'offre reçu par un prestataire.

1. Le prestataire évalue si l'offre est intéressante et, si c'est le cas, il demande à un agent s'il est disponible pour réaliser la tâche. Le choix de cet agent n'est pas traité ici. L'intérêt de l'offre repose sur le calcul de la marge que le prestataire a pour la réalisation de la tâche ; celle-ci doit être positive.
2. Si l'agent contacté est disponible, alors le prestataire crée une offre de service à partir de l'appel d'offre en cours de traitement.
3. Le prestataire soumet ensuite son offre de service au gestionnaire émetteur de l'appel d'offre.

Représentez cet extrait de la deuxième étape du protocole sous forme d'un diagramme de séquence d'UML.

Choix du prestataire

Dans cette étape, le gestionnaire dépouille les offres de service qu'il a reçues et choisit le prestataire.

1. Le gestionnaire effectue l'opération de dépouiller les offres qui lui permet de sélectionner une offre convenable (selon un algorithme pas détaillé ici).
2. Il envoie un message au prestataire émetteur de l'ordre puis affecte le prestataire au demandeur, en précisant la demande initiale et la référence du prestataire.

Représentez cette dernière étape du protocole sous forme d'un diagramme de séquence d'UML.

Propriétés des agents

Quelles propriétés des différentes classes d'agent cette modélisation vous a-t'elle permis de découvrir ? Représentez-les sous forme d'un diagramme de classes.

7.4 Client mail

On s'interresse ici à la modélisation de la réalisation d'un envoi de message par l'application *client mail* à l'initiative de l'utilisateur.

1. L'utilisateur créé un nouveau message.
2. Une fois le message créé, il peut lui ajouter des destinataires (un nombre compris en 1 et n) et saisir le corps du message.
3. L'utilisateur peut ensuite demander d'expédier le message en indiquant le compte de messagerie à utiliser ; c'est ce dernier qui se charge de l'envoi du message.
4. Le compte de messagerie se charge de faire appel au service de transfert de message du `ServeurEmission` ; ce dernier envoie le message à chacun des destinataires et retourne un indicateur permettant de savoir si le transfert s'est passé correctement.
5. En cas d'erreur de transfert, le compte de messagerie interrompt l'envoi et un message est envoyé à l'utilisateur (en mode asynchrone) pour l'en avertir.

Représentez la réalisation de cette fonctionnalité sous la forme d'un diagramme de communication UML.

7.5 Grapher

On s'intéresse ici à la modélisation de deux comportements du grapher mettant en œuvre plusieurs des objets qui lui sont associés. Reportez-vous à l'exercice sur le modèle de classe du grapher.

1. Création du grapher (partiel), impact sur l'axe Ox :
   (a) la création du grapher consiste à créer son axe Ox en lui fournissant une position `p: Base3D` ;
   (b) ensuite l'axe est attaché au grapher (opération `AttachTo`) ;
   (c) enfin, on définit le domaine de l'axe, création d'un domaine en renseignant ses attributs et appel de l'opération `defineDomain(in d: Domain)` sur l'axe Ox.
2. Redéfinition du domaine visible (partiel) :
   il faut définir, pour chaque surface du grapher, le nouveau domaine visible par l'opération `setVisible(in: d Domain)` qui retourne le domaine formé par l'intersection du domaine

courant et de `d`. Si ce domaine est non vide (opération `isEmpty(): boolean`), alors on affiche la surface (opération `draw()` de `Surface`).

Représentez chacune de ces fonctionalités sous forme d'un des modèles des interactions d'UML.

7.6 SimLife

On s'intéresse ici à la modélisation du comportement d'un simulateur. Celui-ci permet de simuler la vie d'animaux dans leur environnement. Le logiciel comprend les classes suivantes.

`Simulator` : le simulateur qui, à chaque pas de son exécution, fait évoluer l'environnement et fait vivre les animaux.
- ◇ `+step()` : exécution d'un pas de simulation
- ◇ `#randomPop(out pop Animal[*]): integer` : obtention de la population des animaux, rangés aléatoirement ; cette opération retourne le nombre d'animaux présents.
- ◇ `+endOfSim(): boolean` UMLConstraintquery : évalue la condition de fin de la simulation.

`Environment` : l'environnement dans lequel évoluent les animaux.
- ◇ `+evolve()` : fait évoluer l'environnement à chaque pas de simulation.

`Animal` : animal dont on simule le comportement
- ◇ `+live()` : comportement de l'animal qui s'exécute à chaque pas de simulation.

Le fonctionnement du simulateur est le suivant.

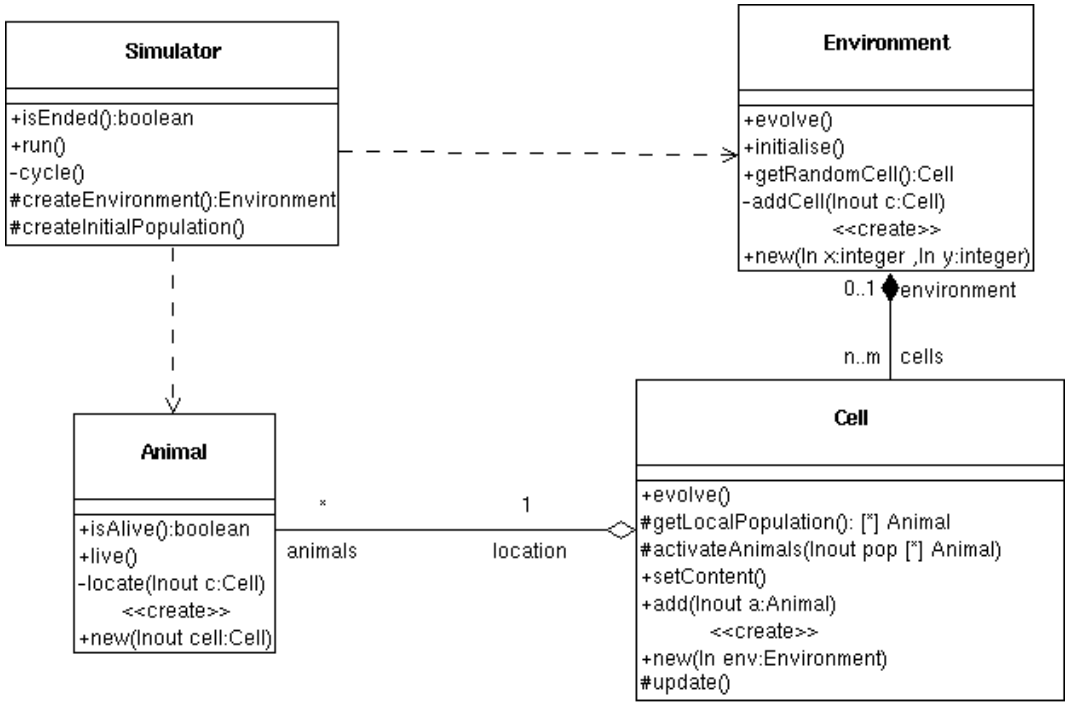
1. Tant que la condition de fin de simulation n'est pas vérifiée, le simulateur effectue un pas de simulation.
2. A chaque pas de simulation, le simulateur commence par faire évoluer l'environnement.
3. Ensuite, il récupère la population d'animaux.
4. Pour finir, il fait vivre chaque animal de cette population.

Représentez chacune de ces fonctionalités sous forme d'un des modèles des interactions d'UML.

7.7 SimLife 2

L'étude porte sur une partie d'un logiciel de simulation de type vie artificielle. On simule la vie d'animaux qui évoluent dans un environnement hétérogène.

On considère que le modèle de classes a déjà été construit. La figure ci-dessous en est un extrait.



1. L'environnement (classe `Environment`) est discrétisé en cellules (classe `Cell`) ; il s'agit d'un rectangle de $n \times m$ cellules.
2. À chaque cellule est associé un type de sol : `Grass`, `Tree`, `Rock`, `Water`...
3. La simulation est cyclique : à chaque cycle, le simulateur fait évoluer l'environnement et on fait vivre chaque animal qui le peuple.

Création de l'environnement

Lorsque l'utilisateur lance le simulateur, celui-ci commence par créer un environnement en précisant ses dimensions ($n \times m$). La création de l'environnement consiste à créer toutes les cellules et à les ajouter à l'environnement. L'environnement crée des cellules dont le type est tiré aléatoirement.

> Représentez ce fonctionnement par un diagramme de communication et complémenter votre diagramme de classe en conséquence.

Création d'une population initiale d'animaux

La création de la population initiale consiste à créer p_E instances de chaque spécialisation E de la classe `Animal` et de les placer aléatoirement dans les cellules. Pour cela, le simulateur demande à l'environnement de lui fournir une cellule tirée aléatoirement. Ensuite, il crée un animal en lui précisant la position à laquelle il se trouve. Cela se fait en appelant l'opération `locate()` de la classe `Animal` qui fait elle-même appel à l'opération `add(inout :a Animal)` d'`Environment`.

> Représentez ce fonctionnement par un diagramme de communication et complémenter votre diagramme de classe en conséquence.

Un cycle de simulation

À chaque cycle, tant que la condition de fin de simulation n'est pas vérifiée, le simulateur effectue l'opération `cycle()` qui consiste à faire évoluer l'environnement et faire vivre les animaux.

L'évolution de l'environnement consiste à faire évoluer chaque cellule, ce qui permet – par exemple – de faire pousser l'herbe.

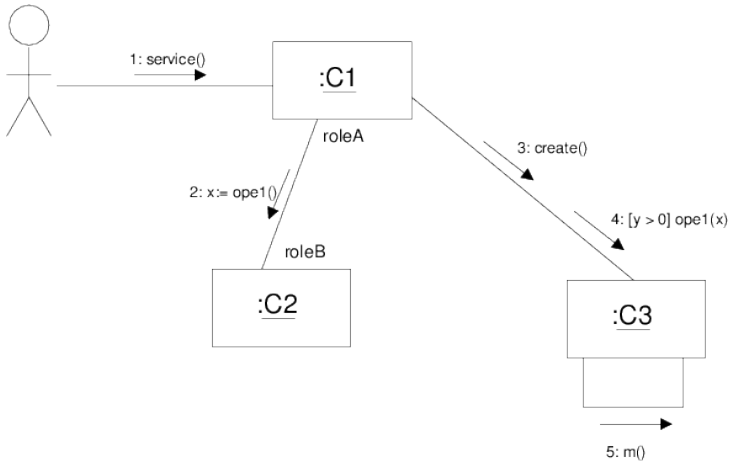
L'activation d'un animal consiste à lui faire exécuter l'opération `live()` à condition qu'il soit vivant, ce qui est évaluable par appel de `isAlive(): boolean {query}`. Si le comportement de l'animal fait qu'il se déplace, il faut éventuellement mettre à jour sa localisation dans l'environnement.

> Représentez ce fonctionnement par un diagramme de communication et complémenter votre diagramme de classe en conséquence.

7.8 Diagrammes de communication => classes

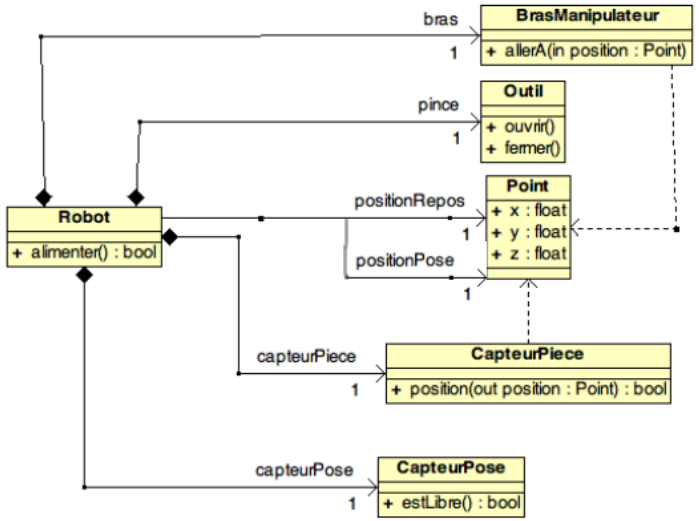
Des interactions entre objets sont représentées sur le diagramme de communication ci-dessous.

Construire le diagramme de classes correspondant.



7.9 Commande d'un robot

L'étude porte ici sur la commande d'un robot comprenant un bras manipulateur et équipé d'une pince pour la prise de pièces (une à la fois). La mission du robot est de prendre des pièces situées en des positions quelconques et de les placer en un endroit prédéfini, appel à point de pose. Le robot est équipé d'un capteur qui lui permet de localiser la pièce à prendre en (x, y, z) et d'un capteur lui permettant de savoir si le point de pose est libre ou non.



Lorsque le robot reçoit un ordre d'alimentation, c'est-à-dire placer une pièce point de pose, il vérifie d'abord que celui-ci est libre avant de réaliser sa mission. La première partie de la mission consiste a localiser la pièce a prendre, ensuite a positionner le bras de façon a attraper a pièce, et finalement à fermer la pince pour saisir la pièce. La deuxième partie est la pose de la pièce : positionnement du bras au point de pose, ouverture de la pince pour déposer la pièce. Pour finir, le robot positionne le bras en position de repos.

Modélisez ces interactions entre le système de commande, le robot et ses différents composants sous forme d'un modèle d'interaction UML.

Solutions Labo

8 Diagramme de classes

Sommaire

8.1 Classes

8.2 Associations

8.2.1 Exo Vol : 5.2.2

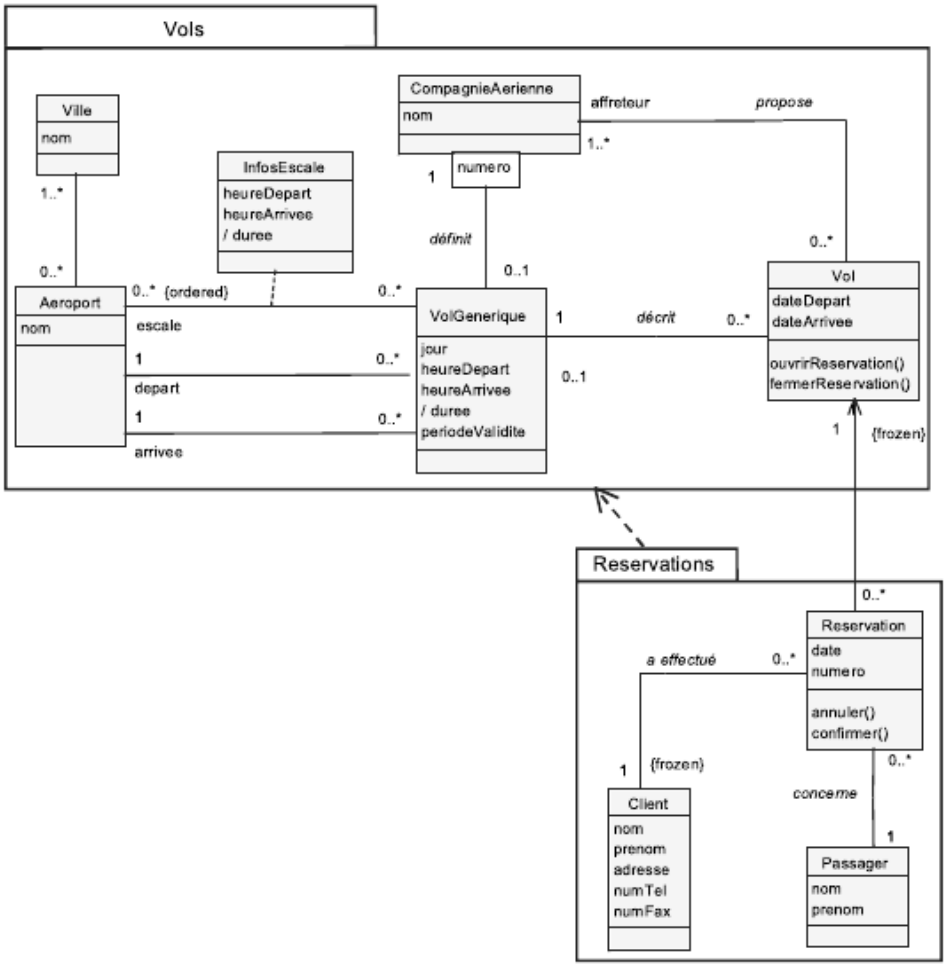


Figure 8.1.

8.3 Relations

8.3.1 Exo Module Pedagogique : 5.3.1

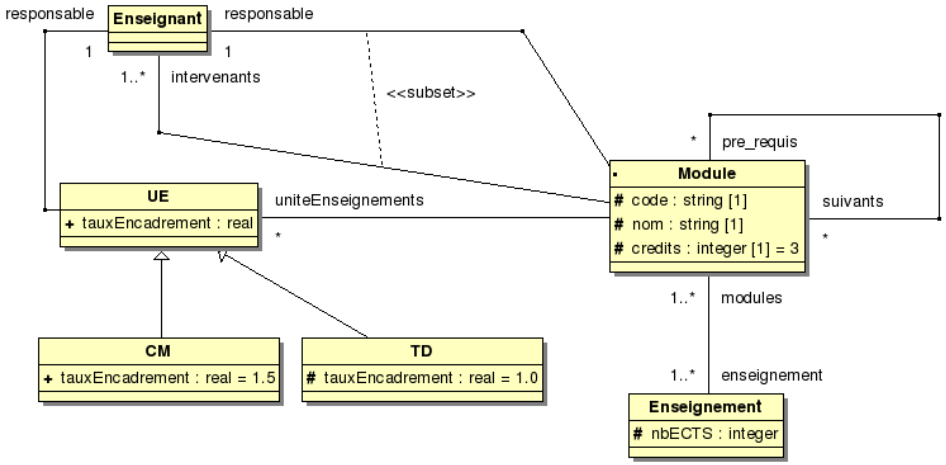


Figure 8.2.

8.3.2 Exo Gestion Projet : 5.3.2

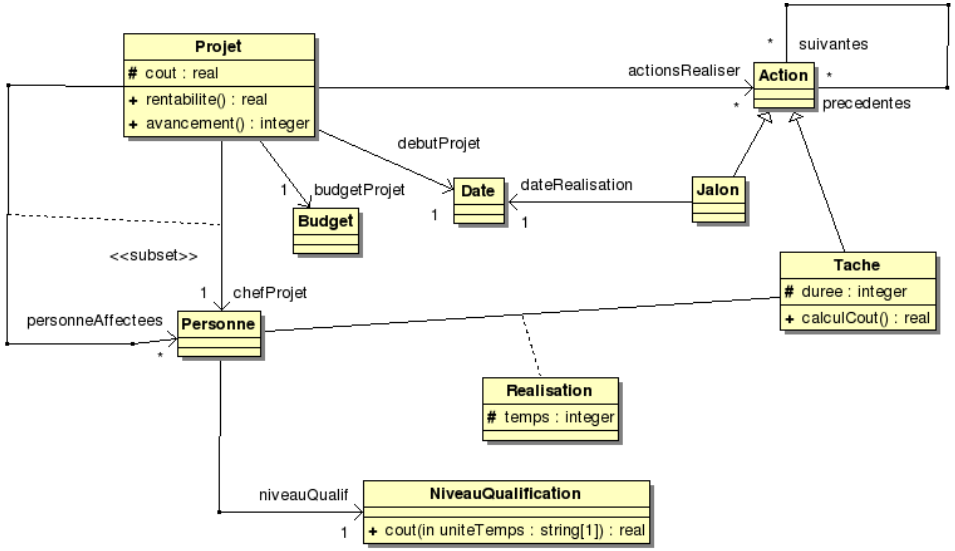


Figure 8.3.

9 Diagramme de cas d'utilisation

10 Diagramme d'interactions