

COURS 2 : UML diagrammes de cas d'utilisation et de relations entre classes

DIAGRAMMES DE CAS D'UTILISATION	3
1- Introduction	3
2- Principaux éléments des diagrammes de cas d'utilisation	3
3- Les acteurs	4
4- Relations liant les acteurs	5
4.1 Associations entre cas et acteurs	5
4.2 Relations entre acteurs	6
4.3 Relations entre cas d'utilisation.....	7
4.4 Généralisation entre cas d'utilisation	7
4.5 Réutilisation de cas d'utilisation	8
4.6 Décomposition de cas d'utilisation	8
4.7 Synthèse	9
4.8 Description textuelle de cas d'utilisation	9
RELATIONS ENTRE CLASSES ET IMPLEMENTATION EN JAVA	10
1. Relation de dépendance.....	10
Exercice 1	11
2. Relation d'association	12
Exercice 2	13
Exercice 3	13

Exercice 4	14
Exercice 5	15
Exercice 6	16
Exercice 7	17
Exercice 8	18
3. Agrégation et composition	18
Exercice 9	19
Exercice 10	20
4. Navigabilité	22
Exercice 11	22
5. Héritage : généralisation et spécialisation	23
Exercice 12	24
Exercice 13	24
Exercice 14	25
Exercice 15	25



1- Introduction

La maîtrise d'ouvrage et les utilisateurs ne sont pas des informaticiens. Il leur faut donc un moyen simple d'exprimer leurs besoins. C'est précisément le rôle des diagrammes de cas d'utilisation qui permettent de recueillir, d'analyser et d'organiser les besoins, et de recenser les grandes fonctionnalités d'un système. Il s'agit donc de la première étape UML d'analyse d'un système.

Qu'est-ce que l'acronyme **UML** : « *Le langage UML (Unified Modeling Language, ou langage de modélisation unifié) a été pensé pour être un langage de modélisation visuelle commun, et riche sémantiquement et syntaxiquement.* »

Un diagramme de cas d'utilisation capture le comportement d'un système, d'un sous-système, d'une classe ou d'un composant tel qu'un utilisateur extérieur le voit. Il scinde la fonctionnalité du système en unités cohérentes, les cas d'utilisation, ayant un sens pour les acteurs. Les cas d'utilisation permettent d'exprimer le besoin des utilisateurs d'un système, ils sont donc une vision orientée utilisateur de ce besoin au contraire d'une vision informatique.

Il ne faut pas négliger cette première étape pour produire un logiciel conforme aux attentes des utilisateurs. Pour élaborer les cas d'utilisation, il faut se fonder sur des entretiens avec les utilisateurs.

Source : [UML 2 - de l'apprentissage à la pratique](#)

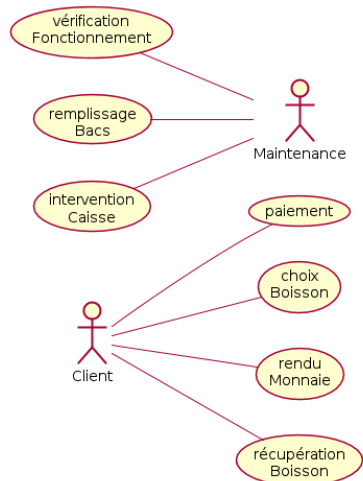
2- Principaux éléments des diagrammes de cas d'utilisation

Avant tout développement, il convient de répondre à la question : "A quoi va servir le logiciel ?" sous peine de fournir des efforts considérables en vain.


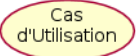
En UML, on établit des *Diagrammes de Cas d'Utilisation* pour répondre à cette question.

Source : [UML Cours 1 : Diagrammes de cas d'utilisation](#)

Exemple d'un diagramme de cas d'utilisation :



Principaux éléments :

- Acteurs (bonhommes) :  entité extérieure au système modélisé, et qui interagit directement avec lui.
- Cas d'utilisation (ellipses) :  : service rendu à un acteur, c'est une fonctionnalité de son point de vue.

3- Les acteurs

- **Les principaux acteurs sont les utilisateurs du système**
- **En plus des utilisateurs, les acteurs peuvent être non humains :**
 - Des logiciels déjà disponibles à intégrer dans le projet ;
 - Des systèmes informatiques externes au système mais qui interagissent avec lui ;
 - Tout élément extérieur au système et avec lequel il interagit.

Pour identifier les acteurs, on se fonde sur les frontières du système.

➤ **Rôles et personnes physiques.**

- Un acteur correspond à un rôle, pas à une personne physique.
- Une même personne physique peut être représentée par plusieurs acteurs si elle a plusieurs rôles.
- Si plusieurs personnes jouent le même rôle vis-à-vis du système, elles seront représentées par un seul acteur.

Exemples :

Pierre, Paul ou Jacques sont tous les Clients du point de vue d'un distributeur automatique

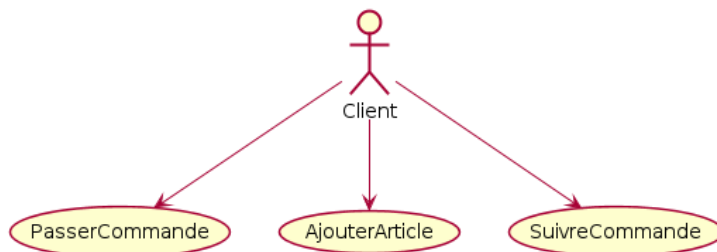
En tant que webmestre, Paul a certains droits sur son site, mais il peut également s'identifier en tant qu'utilisateur quelconque, sous une autre identité

4- Relations liant les acteurs

4.1 Associations entre cas et acteurs

Les acteurs demandant des services aux systèmes, ils sont le plus souvent à l'initiative des échanges avec le système : ils sont dits *acteurs primaires*.

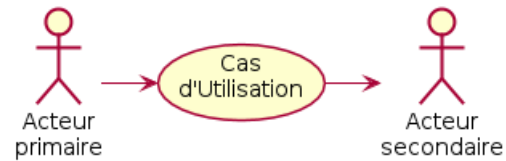
Lorsqu'ils sont sollicités par le système (dans le cas de serveurs externes par exemple), ils sont dits *acteurs secondaires*.



On représente une association entre un acteur et un cas d'utilisation par une ligne pleine.



Un acteur est souvent associé à plusieurs cas d'utilisation



4.2 Relations entre acteurs

Il n'y a qu'un seul type de relation possible entre acteurs : la relation de *généralisation*.

Il y a généralisation entre un cas A et un cas B lorsqu'on peut dire : *A est une sorte de B*. Exemple :

- Un directeur est une sorte de commercial : il peut faire avec le système tout ce que peut faire un commercial, plus d'autres choses



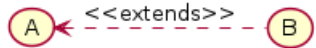
4.3 Relations entre cas d'utilisation

Types de relations possibles

- *Inclusion* : B est une partie obligatoire de A et on lit *A inclut B* (dans le sens de la flèche).



- *Extension* : B est une partie optionnelle de A et on lit *B étend A* (dans le sens de la flèche).



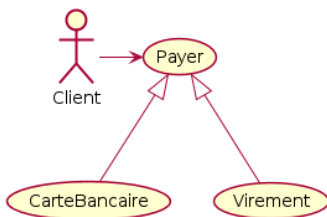
- *Généralisation* : le cas A est une généralisation du cas B et on lit *B est une sorte de A*.



Les flèches en pointillés dénotent en fait une relation de *dépendance*, et les mentions *includes* et *extends* sont des *stéréotypes* et à ce titre placés entre guillemets.

4.4 Généralisation entre cas d'utilisation

Cette relation de généralisation/spécialisation est présente dans la plupart des diagrammes UML et se traduit par le concept d'héritage dans les langages de programmation orientés objet.

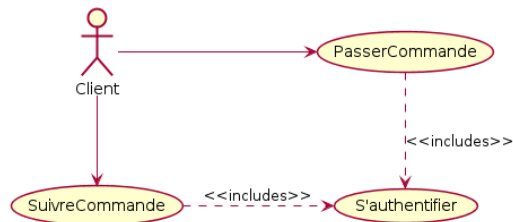


Cet héritage signifie que les éléments spécifiques *héritent* de tout ce qui caractérise l'élément général : - Les associations avec des acteurs - Les relations de dépendance - Les héritages déjà existant, dans lesquels l'élément général joue le rôle d'élément plus spécifique

4.5 Réutilisation de cas d'utilisation

Les relations d'inclusion et d'extension permettent d'isoler un service réutilisable comme partie de plusieurs autres cas d'utilisation :

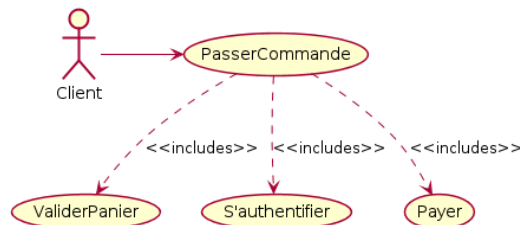
- On parle alors de *réutilisation*.
- Le code développé pour implémenter le cas d'utilisation réutilisé est d'emblée identifié comme ne devant être développé qu'une seule fois, puis réutilisé.



4.6 Décomposition de cas d'utilisation

Un cas d'utilisation ne doit jamais se réduire à une seule action : il doit occasionner des traitements d'une complexité minimale.

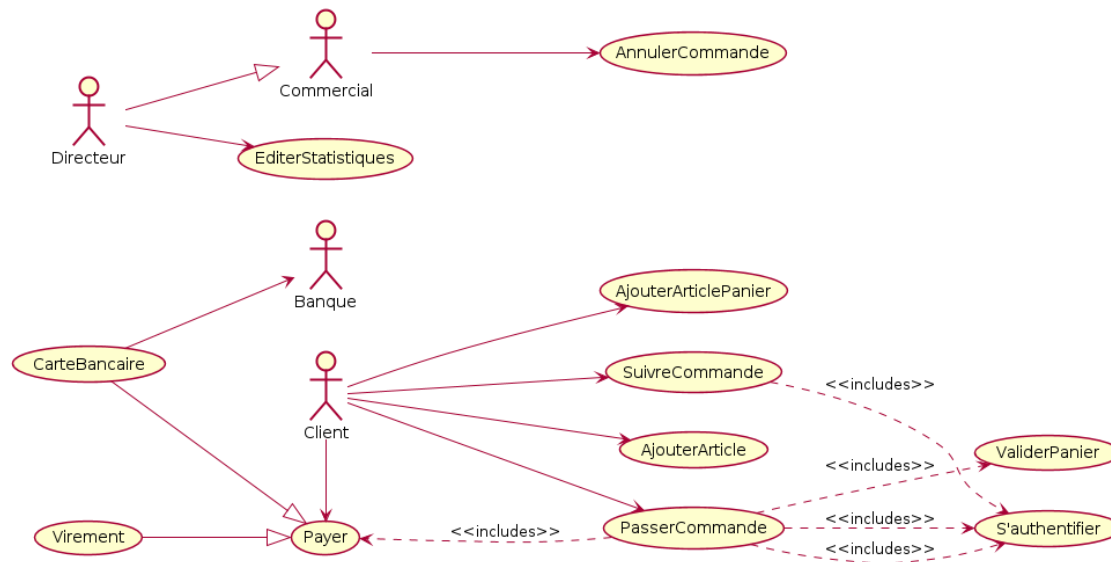
Toutefois, il peut arriver qu'un cas d'utilisation recouvre un ensemble très important d'échanges et de traitements. Dans ce cas, on peut utiliser les relations d'inclusion et d'extension.



Attention : ne jamais décomposer en première analyse, mais lorsqu'on a suffisamment avancé dans la conception ou le codage pour être certain que la décomposition est utile.

4.7 Synthèse

Exemple de diagramme complet :



4.8 Description textuelle de cas d'utilisation

Un nom ne suffit pas à comprendre le détail de ce que recouvre un cas d'utilisation.

Il est donc nécessaire d'adjoindre à chaque cas d'utilisation une description détaillée. Cette description est parfois textuelle et composée de plusieurs rubriques dont les plus importantes sont :

- Le *scénario nominal* : enchaînement d'actions typiques dans le cas où les choses se passent comme prévu.
- Les *enchaînements alternatifs* : enchaînements dans des cas particuliers.

RELATIONS ENTRE CLASSES ET IMPLEMENTATION EN JAVA

10

Les classes ne valent que par les relations qu'elles tissent entre elles. Cette phase est décisive dans l'élaboration du diagramme de classes selon le modèle UML (Unified Modeling Language) :

- [https://fr.wikipedia.org/wiki/UML_\(informatique\)](https://fr.wikipedia.org/wiki/UML_(informatique))
- <https://openclassrooms.com/fr/courses/2035826-debutez-lanalyse-logicielle-avec-uml/2035851-uml-c-est-quoi>
- <https://www.lucidchart.com/pages/fr/quest-ce-que-le-langage-de-modelisation-unifie>
- https://fr.wikipedia.org/wiki/Diagramme_de_classes
- <https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-classes>

Mais c'est à partir des interactions entre les objets que doivent se dessiner les relations entre classes.

1. Relation de dépendance

- **C'est la relation la moins contraignante entre les classes.**
- **Elle est unidirectionnelle et définit une relation d'utilisation.**
- **Dans une collaboration d'objets, cela correspond à l'envoi de message paramétré par un objet d'une autre classe.**

Exemple :

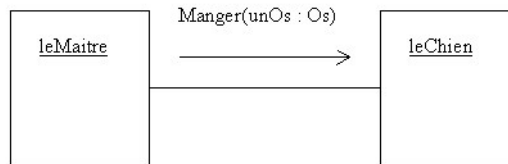


Diagramme d'objets

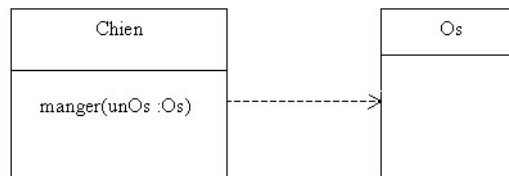


Diagramme de classes partiel

Exercice 1 : D'après le diagramme de classes ci-dessus, implémenter en Java ses 3 classes, en tenant compte des relations et de la méthode *manger()*.



Comment traduire en Java qu'un objet *leMaitre* de la classe *Maitre* donne à « manger » un objet *unOs* de la classe *Os* à un objet *leChien* de la classe *Chien* ? De même, comment traduire en Java que cet objet *unChien* mange l'objet *unOs* ?

En fonction de vos réponses, implémentez en Java les propriétés de ces 3 classes en y incluant leurs propriétés suivantes nécessaires :

- Leurs attributs tous **public** : dans la classe *Chien*, par exemple sa race
- Leur constructeur pour initialiser le.s attribut.s
- La méthode *manger()*, en respect des diagrammes d'objets et de classes ci-dessus
- Le programme **main** : dans la classe *Maitre* instancie les 3 objets *leMaitre*, *leChien* et *unOs*, appeler la méthode *manger()* entre les objets *leChien* et *unOs* d'un. Puis il affiche les attributs respectifs de ces 3 objets. Testez votre programme.

2. Relation d'association

Soit la situation suivante où différents articles sont stockés dans différents entrepôts.

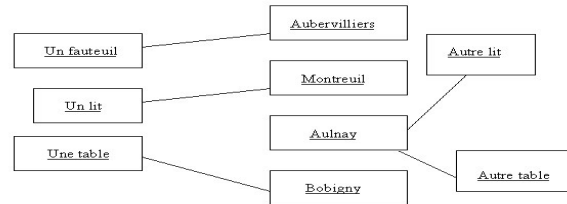


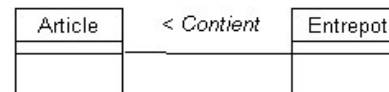
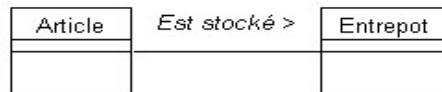
Diagramme d'objets

Remarque : si la classe de chaque objet est clairement identifiée, nous pouvons représenter des objets anonymes, comme instance de classe ; dans ce cas le nom de l'objet est remplacé par une instance. Le diagramme suivant a la même sémantique.

➤ **Les 2 associations suivantes sont équivalentes car elle concerne les mêmes couples d'objets.**

Exemple : La première association (à gauche) a un nom "Est stocké", le symbole ">" indique seulement un sens de lecture de la classe *Article* vers la classe *Entrepot*, pour indiquer qu'un objet de la classe *Article* « est stocké » dans un objet de *Entrepot*. La seconde association (à droite) a un nom "contient", le symbole "<" indique seulement un sens de lecture de la classe *Entrepot* vers la classe *Article*, pour indiquer qu'un objet de *Entrepot* contient des objets de la classe *Article*.

Diagrammes de classes



Remarque : nous utilisons une forme verbale pour qualifier le nom de l'association.

Exercice 2 : D'après le diagramme de classes à gauche ci-dessus, indiquant qu'un objet de la classe *Article* « est stocké » dans un objet de *Entrepot*, implémenter en Java ses 2 classes *Article* et *Entrepot*, avec au moins un attribut et une méthode cohérents par classe.



Comment traduire en Java qu'un objet de la classe *Article* « est stocké » dans un objet de la classe *Entrepot* ?

En fonction de vos réponses, implémentez en Java les propriétés de ces 2 classes en y incluant leurs propriétés suivantes nécessaires :

- Leurs.s attribut.s tous **public** : dans la classe *Article*, par exemple son numéro d'article et son objet stocké dans la classe *Entrepot*.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- La méthode *stocker()* dans la classe *Article*, en respect du diagramme de classes ci-dessus : stocke un objet de *Entrepot* en paramètre de l'objet référencé de la classe *Article* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)).
- Le programme **main** : dans la classe *Entrepot* instancier les objets des classes *Entrepot* et *Article*, appeler la méthode *stocker()* entre ces objets. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

Exercice 3 : D'après le diagramme de classes ci-dessus entre les classes *Article* et *Entrepot*, reprendre l'**exercice 2** précédent en Java pour indiquer qu'un objet de classe *Entrepot* « contient » des objets de la classes *Article*, et inversement tout objet de la classe *Article* «est stocké » dans un objet de la classe *Entrepot*. u moins un attribut et une méthode cohérents par classe.



Que faut-il ajouter dans le diagramme ci-dessus pour indiquer qu'un entrepôt peut être « gardien » de plusieurs articles (voir plus bas le chapitre [Les associations comportent des cardinalités](#)) ? Comment traduire en Java qu'un objet de la classe *Entrepot* « est gardien » de plusieurs objets de la classe *Article* ?

En fonction de vos réponses, implémentez en Java les propriétés de ces 2 classes en y incluant leurs propriétés suivantes nécessaires :

- Leurs.s attribut.s tous **public** : dans la classe *Entrepot*, par exemple les objets de la classe *Article*.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- La méthode *garder()* dans la classe *Entrepot* S: ajoute un objet de la classe *Article* en paramètre de l'objet référencé de la classe *Entrepot* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)), dans la limite à vérifier de 3 articles par entrepôt.
- Dans le programme **main** : instancier les objets des classes, appeler la méthode *garder()* de l'objet de la classe *Entrepot*, dans la limite de 3 articles par entrepôt. Puis afficher les attributs respectifs des objets des classes *Article* et *Entrepot*. Testez votre programme.

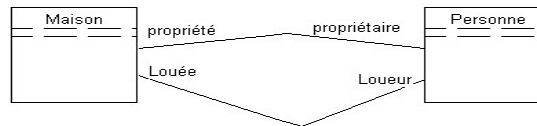


L'association porte un nom



Au niveau des classes, nous pouvons construire une association

- ✓ Une association relie des classes dont les instances sont sémantiquement liées.
- ✓ Une association ne sous-entend aucune hiérarchie dans la relation.
- ✓ Le nom de l'association traduit la nature du lien entre les objets de classes : " Les lits sont stockés dans l'entrepôt de Montreuil".
- ✓ On peut être amené à préciser le rôle que joue chaque classe dans l'association : ci-dessous, il s'agit des rôles de "gardien" côté entrepôt et "gardé" côté article.
- ✓ Mais la notion de rôle prend tout son sens lorsqu'un couple de classe est relié par deux associations sémantiquement distinctes.



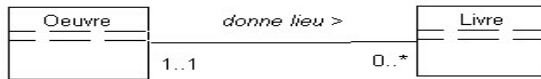
Exercice 4 :  En fonction des explications de cours ci-dessus, expliquez clairement la signification des relations entre les 2 classes *Maison* et *Personne* ?

➤ Les associations comportent des cardinalités

- ✓ Dans UML, on parle de valeurs de cardinalités (multiplicités).

Valeurs de multiplicité :

1..1	Un et un seul
1	Un et un seul
0..1	Zéro ou un
m..n	De m à n
*	De zéro à plusieurs
0..*	De zéro à plusieurs
1..*	Un à plusieurs



Exemple : Une œuvre donne lieu à 0 ou plusieurs livres. Un livre est issu d'une et une seule œuvre.

15

Exercice 5 : D'après le diagramme de classes ci-dessus, indiquant qu'un objet de la classe *Oeuvre* « donne lieu » à des objets de la classe *Livre*, implémenter en Java ses 2 classes *Oeuvre* et *Livre*, avec au moins un attribut et une méthode cohérents par classe.



Comment traduire en Java qu'un objet de la classe *Oeuvre* « donne lieu » à des objets de la classe *Livre*, en tenant compte des cardinalités ?

Inversement, comment indiquer qu'un objet de la classe *Livre* est issu d'un seul objet de la classe *Oeuvre* ?

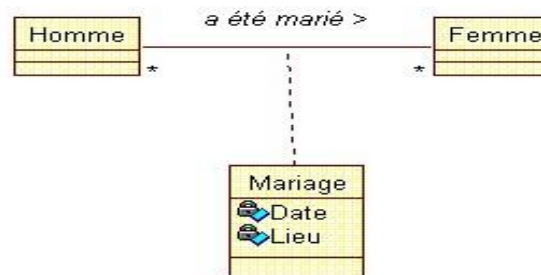
En fonction de vos réponses, implémentez en Java les propriétés de ces 2 classes en y incluant leurs propriétés suivantes nécessaires :

- Leurs.s attribut.s tous **public** : dans la classe *Entrepot*, Vles objets de la classe *Article*.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- La méthode *garder()* dans la classe *Entrepot* S: ajoute un objet de la classe *Article* en paramètre de l'objet référencé de la classe *Entrepot* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)), dans la limite à vérifier de 3 articles par entrepôt.
- Dans le programme **main**, appeler la méthode *garder()* de l'objet de la classe *Entrepot*, dans la limite de 3 articles par entrepôt. Puis afficher les attributs respectifs des objets des classes *Article* et *Entrepot* Testez votre programme.
- Leurs.s attribut.s tous **public** : par exemple dans la classe *Livre*, son titre
- Leur.s constructeur.s pour initialiser le.s attribut.s
- La méthode *issu()* dans la classe *Livre* : associe l'objet de la classe *Oeuvre* en paramètre à l'objet référencé de la classe *Livre* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)).
- La méthode *donnerlieu()* dans la classe *Oeuvre* : : ajoute un objet de la classe *Livre* en paramètre dans l'objet référencé (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)) de la classe *Oeuvre*, dans la limite à vérifier de 3 livres par œuvre.
- Le programme **main** : dans la classe *Oeuvre* instancier les objets des classes *Livre* et *Oeuvre* , appeler leur méthode dans la limite de 3 livres par oeuvre. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

En Java, une multiplicité supérieure à 1 se traduit par une collection d'objets (tableau, liste etc.)

➤ Classe association porteuse d'attributs

- ✓ L'association entre deux classes peut elle-même posséder des attributs.



- La présence des "cadenas" provient simplement d'un logiciel utilisé pour désigner des attributs privés.
- La classe-association ne porte pas de valeurs de multiplicité. Chaque instance de la classe-association *Mariage* est reliée à un couple d'instances des classes *Homme* et *Femme*.

Exercice 6 :



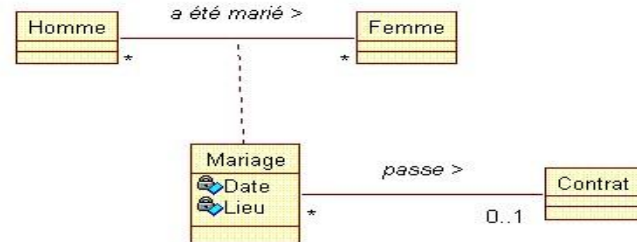
Comment traduisez-vous le diagramme ci-dessus, en tenant compte des classes, de leur relation avec le nom, des cardinalités et des attributs.

En déduire l'implémentation Java avec les propriétés suivantes des 3 classes *Mariage*, *Homme* et *Femme* de ce diagramme :

- Leurs attributs tous **public** : dans la classe *Mariage*, l'exemple a date et lieu de mariage.
- Leur constructeur pour initialiser les attributs.
- La méthode *marié()* dans la classe *Mariage* : ajoute les 2 objets des classes *Homme* et *Femme* en paramètres de l'objet référencé de la classe *Mariage* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)).
- Dans le programme **main**, dans la classe *Mariage* instancier les objets des classes 3 classes, appeler la méthode *marié()* des objets de la classe *Mariage* entre les objets des classes *Homme* et *Femme*. Puis afficher les attributs respectifs de ces objets. Testez votre programme.
- La méthode *donnerlieu()* dans la classe *Œuvre* : ajoute un objet de la classe *Livre* en paramètre dans l'objet référencé (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)) de la classe *Œuvre*, dans la limite à vérifier de 3 livres par œuvre.
- Le programme **main** dans la classe *Œuvre* instancie les objets des classes *Livre* et *Œuvre*, appelle leur méthode dans la limite de 3 livres par œuvre. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

➤ Une classe-association peut participer à d'autres relations.

A partir de l'exemple précédent, imaginons que le mariage soit l'objet d'un contrat de mariage type.



Commentaires : la classe-association *Mariage* peut participer à des relations propres. Chaque mariage donne lieu ou pas à un contrat type.

Exercice 7 :



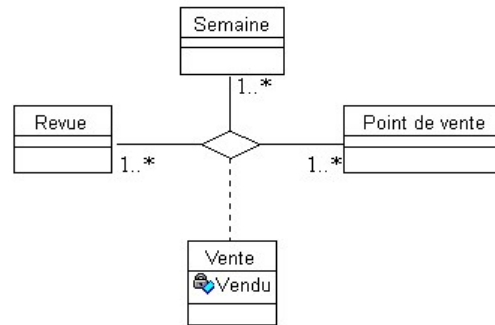
Comment traduisez-vous la relation entre la classe *Mariage* et *Contrat*, en tenant compte des cardinalités ?

En déduire l'implémentation Java avec les propriétés suivantes des classes de ce diagramme :

- Le.s attribut.s **public** des classes *Mariage* et *Contrat* : dans la classe *Contrat*, par exemple le numéro de contrat du mariage.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- La méthode *passer()* dans la classe *Mariage* : ajoute l'objet de la classe *Contrat* en paramètre de l'objet référencé de la classe *Mariage* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)).
- Dans le programme **main** : dans la classe *Mariage* instancier un objet de la classe *Contrat*, appeler la méthode *passer ()* de l'objet de la classe *Mariage*, avec en paramètre l'objet de la classe *Contrat*. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

➤ Association à arité supérieure à 2

Commentaires : Prenons le thème des revues disponibles dans différents points de vente. Imaginons que nous désirions suivre les disponibilités hebdomadaires.



Exercice 8 :



Comment traduisez-vous la relation entre les 4 classes du diagramme ci-dessus, en tenant compte commentaires et des cardinalités précédents ?

Implémentez en Java ces 4 classes, en respect du diagramme et des commentaires précédents, avec les propriétés suivantes :

- Le.s attribut.s **public** de ces 4 classes : dans la classe *Vente*, par exemple le numéro le nombre vendu par objet des classes *Revue*, *PointDeVente* et *Semaine*.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- Les méthodes nécessaires pour chacune des classes du diagramme ci-dessus : par exemple *vendre()* dans la classe *Vente*, indiquant les objets les objets des classes *Revue*, *PointDeVente* et *Semaine* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)) en paramètres de l'objet référencé de la classe *Vente*.
- Dans le programme **main**, : dans la classe *Vente* instancier les objet de ces 4 classes, appeler le.s méthode.s de leur.s objet.s,. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

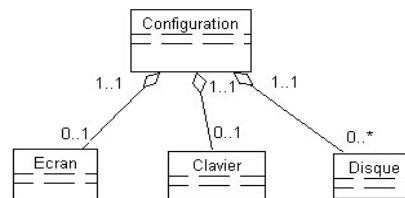
3. Agrégation et composition

Une association relie deux - ou plus - classes, sans induire de rôle particulier pour une classe ; l'association se lie indistinctement dans les deux sens.
L'agrégation et la composition vont modéliser une situation où l'une des classes joue un rôle particulier, l'association n'est plus symétrique.

➤ L'agrégation

- ✓ L'agrégation traduit une relation d'appartenance de l'agrégé (contenu) dans l'agrégat (contenant)
- ✓ Elle ne sous-entend aucune valeur de multiplicité particulière.
- ✓ L'agrégation peut être assimilée à une appartenance faible : la classe d'agrégat contenant et la classe d'agrégé contenu peuvent être indépendante l'une de l'autre, il n'y a pas de lien exclusif entre ces 2 classes.
- ✓ L'agrégation se modélise par un losange côté agrégat.

Commentaires : Soit une configuration constituée d'un certain nombre d'éléments. Une configuration comporte un clavier -ou aucun -, un écran -ou aucun -, et éventuellement plusieurs disques. La destruction d'une configuration n'entraîne pas celle de ses trois composants mais seulement la suppression du lien.



Exercice 9 :



Comment traduisez-vous la relation entre les 4 classes du diagramme ci-dessus, en tenant compte commentaires et des cardinalités précédents ? Quelle différence y voyez-vous avec le diagramme de classes [Association à arité supérieure à 2](#) et de l'exercice 8 précédents ?

Implémentez en Java ces 4 classes, en respect du diagramme et des commentaires précédents, avec les propriétés suivantes :

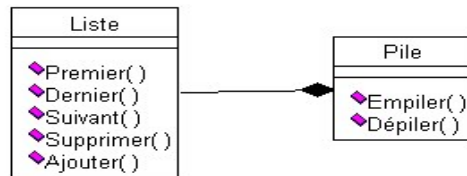
- Le.s attribut.s **public** de ces 4 classes : dans la classe *Configuration*, par exemple son numéro.
- Leur.s constructeur.s pour initialiser le.s attribut.s.
- Les méthodes nécessaires pour chacune des classes du diagramme ci-dessus : par exemple *assembler()* dans la classe *Configuration*, indiquant les objets les objets des classes *Ecran*, *Clavier* et *Disque* (voir [Cours 3 et exercices Semaine 2 Concepts de base](#)) en paramètres de l'objet référencé de la classe *Configuration*.
- Dans le programme **main**, : dans la classe *Configuration* instancier les objet de ces 4 classes, appeler le.s méthode.s de leur.s objet.s,. Puis afficher les attributs respectifs de ces objets. Testez votre programme.

Bien sûr, nous aurons très souvent une multiplicité 1..1 ou 0..1 côté agrégat. L'appartenance est dite faible car l'agrégé pourra participer à d'autres agrégats et son cycle de vie n'est pas subordonné à celui de son agrégat. Dans notre autre exemple, la disparition d'une configuration n'entraîne pas la disparition des périphériques.

➤ La composition

- ✓ Il s'agit d'une appartenance forte.
- ✓ La vie de l'objet composant est liée à celle de son composé.
- ✓ La notion de composant est proche de celle d'attribut, si ce n'est que "l'attribut" est "rehaussé" au rang de classe. On parlera de réification.

Commentaires : Soit une pile constituée d'une liste d'objets. Une pile est constituée d'une liste d'objets, éventuellement vide, ont chaque objet ne peut que s'empiler ou se dépiler en son sommet. Si l'on supprime la liste des objets, toute la pile est supprimée automatiquement.



Exercice 10 :



Comment traduisez-vous la relation entre les 2 classes du diagramme ci-dessus, en tenant compte des commentaires et de leur relation ? Quelle différence y a-t-il entre une agrégation comme dans l'exemple plus haut de l'agrégation de **L'agrégation** et son exemple, et la composition dans l'exemple ci-dessus ?

Contrairement à l'agrégation, que se passe-t-il si on supprime un composé (dans l'exemple ci-dessus, un objet de la classe *Pile*) quel effet cela a-t-il sur l'objet composant (dans l'exemple ci-dessus, un objet de la classe *Liste*) ?

En est-il de même dans l'exemple plus haut de l'agrégation de **L'agrégation** : si on supprime l'objet composant de la classe *Configuration*, supprime-t-on ses objets composés des classes *Ecran*, *Clavier* et *Disque* ?

Implémentez en Java ces 2 classes, en respect du diagramme et des commentaires précédents, avec les propriétés suivantes :

- Dans la classe *Liste*, ses attributs **public** : par exemple, un tableau d'objets de la classe *Livre* (voir **exercice 5**) limité à 3 (voir chapitre [3.12 Tableaux](#) dans le lien [Cours 1 et exercices Semaine 1 Introduction](#)), le nombre maximum de d'objets du tableau et la *position* du dernier objet de la liste initialisé à **-1** (valeur si pas d'objet dans le tableau de la liste)
- Leur.s constructeur.s pour initialiser le.s attribut.s.

- Dans la classe *Liste*, implémenter les méthodes telles que mentionnées dans le diagramme des classes ci-dessus :
 - o *Premier()* sans paramètre. Cette méthode initialise sa *position* à **0** (position du premier objet).
 - o *Dernier()* sans paramètre. Cette méthode initialise sa *position* à **0** si la liste est vide sinon elle initialise la *position* à la taille du tableau.
 - o *Suivant()* sans paramètre, Cette méthode retourne null (objet vide) si le tableau d'objets est vide (*position* à **-1**) ou si la *position* est déjà dans la limite de 3 est atteinte dans le tableau sinon elle incrémente la *position* et retourne l'objet suivant dans le tableau de la liste.
 - o *Supprimer()* sans paramètre. Cette méthode retourne null (objet vide) si le tableau d'objets est vide (*position* à **-1**), sinon elle décrémente la *position* et retourne l'objet d'objet précédent dans le tableau de la liste.
 - o *Ajouter()* avec en paramètre l'objet. Cette méthode ajoute l'objet en paramètre à la *position* du tableau de la liste et incrémente la position.

Dans la classe *Pile*, implémenter les méthodes suivantes comme mentionnées dans le diagramme des classes ci-dessus :

- o *Empiler()* avec en paramètre un objet (ici de la classe *Livre*). Cette méthode appelle les méthodes *Dernier()* et *Ajouter()* de la classe *Liste* pour ajouter un objet (ici de la classe *Livre*) au dernier élément de l'objet de la classe *Liste*.
- o *Depiler()* sans paramètre. Cette méthode appelle les méthodes *Dernier()* et *Supprimer()* de la classe *Liste* pour accéder au dernier objet (ici de la classe *Livre*) au dernier élément de l'objet de la classe *Liste* et retourner cet objet.

Dans une autre classe, implémenter le programme du **main** avec les instructions suivantes à tester :

- o Instancier les objets des classe *Livre* (déjà instancié à l'**exercice 5**), *Liste* et *Pile*.
- o Appeler successivement les méthodes *Empiler()* et *Depiler()* pour vérifier tous les cas possibles de pile avant après empilement et dépilement : pile vide et pile saturée, en affichante les valeurs d'attributs des livres après empilement et dépilement .

Commentaires :

- La composition se modélise par un losange noir côté composé.
- Une application contient de 0 à n fenêtres qui contiennent de 0 à n boutons.
- La fermeture de l'application entraîne la destruction des fenêtres qui entraîne la destruction des boutons.
- La non-présence des valeurs de multiplicités est synonyme de 1..1.

Règles :

- Un composant ne peut appartenir à un moment donné qu'à un composé.
- La cardinalité ne peut être que de 1 maximum côté composé.
- La suppression du composé entraîne celle du composant.

4. Navigabilité

Dans une association, on peut en général naviguer d'une classe à l'autre. Il est parfois nécessaire à certaines étapes de la conception d'indiquer que la navigabilité n'est plus bidirectionnelle. Le parcours de l'association privilégie un sens particulier de navigation. Par exemple dans une gestion de prêts de livre, on peut concevoir que la bibliothèque connaisse ses livres, mais il ne semble pas nécessaire à chaque livre de connaître la bibliothèque. Nous pouvons interpréter ce choix par une navigabilité restreinte.



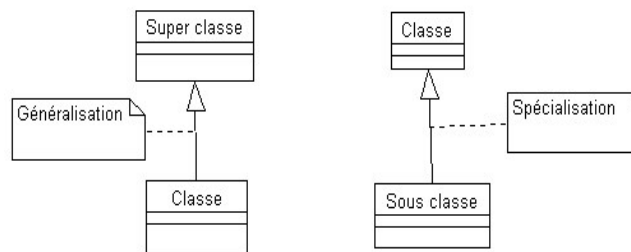
Navigabilité restreinte : de la bibliothèque vers les livres.

Exercice 11 :  En fonction des explications sur la navigabilité ci-dessus, expliquez clairement la signification de la relation entre les 2 classes *Bibliothèque* et *Livre*, en tenant compte des cardinalités ?

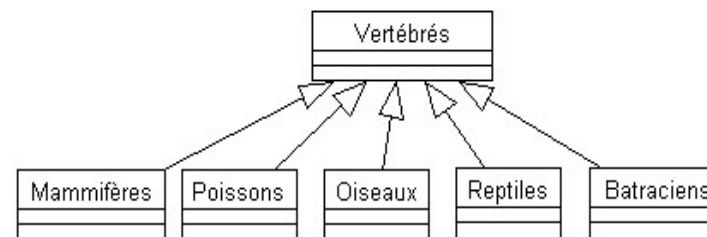
5. Héritage : généralisation et spécialisation

Cette notion est très importante dans le monde objet. Son implémentation sous forme d'héritage donne tout son sens au concept de réutilisation.

Nous avons mis en œuvre un premier niveau de classement en regroupant les objets en classes, nous franchissons là un premier niveau d'abstraction. La généralisation nous propose d'en franchir un second niveau.



Le concept de généralisation est largement utilisé dans le domaine scientifique afin de modéliser un classement hiérarchique. Classement des espèces chez les biologistes, classement des végétaux chez les naturalistes, classement des objets célestes pour les physiciens astronomes.



La spécialisation s'applique lorsqu'une classe affine les propriétés -attributs et comportements- d'une autre classe. Les termes de généralisation et spécialisation s'appliquent à un même type de relation entre classes ; on peut employer l'un ou l'autre selon le sens de lecture.

- L'héritage est un mécanisme fondamentale de Java (et de l'objet)
 - ✓ Permet de spécialiser une classe existante, sans modifier le code de l'existante.
- On l'utilise chaque fois qu'on déclare une classe
 - ✓ En Java toute classe est par défaut une sous-classe de la classe **java.lang.Object**.
- On déclare une sous-classe d'une classe existante (autre que **Object**) par moyen du mot-clé **extends**.

Syntaxe partielle : visibilité **class** nomSousClasse **extends** nomSuperClasse { corps }

Exercice 12 : Implémenter en Java la classe *Vertebres* du diagramme de classes ci-dessus avec les propriétés suivantes :

- Au moins un attribut commun à tous les vertébrés
- 2 constructeurs : l'un par défaut sans paramètre, l'autre avec au moins paramètre pour initialiser le.s attribut.s
- Une méthode qui affiche le.s attribut.s

➤ Initialisation de la superclasse

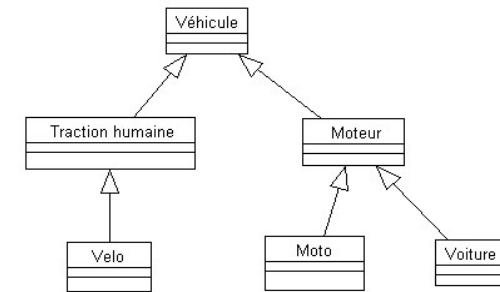
- ✓ Puisqu'une classe hérite les membres de sa superclasse, Java appelle le constructeur de la superclasse pour s'assurer que ces membres sont correctement initialisés.
- ✓ A l'appel du constructeur d'une classe, Java appelle automatiquement le constructeur par défaut de la superclasse.
- ✓ Il est possible d'appeler explicitement les constructeurs de la super-classe avec le mot-clé **super(...)**, ainsi que les méthodes de la super-classe avec le mot clé **super.méthode(...)** :
 - ❖ Doit être la première instruction du constructeur ou de la méthode
 - ❖ Une seule utilisation par constructeur ou méthode
 - ❖ Annule l'appel automatique de **super()**

Exercice 13 : Compléter le code Java précédent de l'**exercice 12** avec la classe *Mammifere* qui hérite de la classe *Vertebres*, en définissant les propriétés suivantes :

- Au moins un attribut commun à tout mammifère
- Constructeurs et la méthode d'affichage qui héritent de ceux de la super-classe *Vertebres*.
- Le programme **main** doit instancier 2 objets de la classe *Vertebres* (pour chacun des 2 constructeurs) et 1 objet de la classe *Mammifere*, puis doit appeler la méthode d'affichage pour les 3 objets.

Autre exemple :

- Les véhicules à traction humaine et à moteur sont des sortes de véhicules. Elles en précisent les propriétés sur certains points, de même pour le vélo qui spécialise les véhicules à traction humaine ou pour les motos et les voitures.
- Les relations de généralisation sont transitives ; une voiture est une sorte de véhicule à moteur mais aussi une sorte de véhicule.
- Il n'y a pas de valeur de multiplicité explicite, par principe il s'agit d'un et un seul dans les deux sens.
 - Une moto ne peut être voiture. La contrainte par défaut est l'exclusion d'objets communs. Nous y reviendrons plus loin.
- En remontant la lecture du diagramme, on peut dire que les propriétés communes des motos et des voitures sont factorisées dans les véhicules à moteur.



Par un raccourci de langage, certains assimilent généralisation et héritage ; ceci devra être évité car l'héritage est une technique supportée par la plupart des langages objets. L'héritage relève de l'implémentation, la généralisation de l'analyse.

Mot clé **protected**

- ✓ Permet de cacher des opérations du monde au sens large, tout en permettant l'accès par les classes dérivées.

Exercice 14 : Implémenter une classe *Maman* qui définit un attribut privé entier, un constructeur qui initialise cet attribut avec le paramètre, puis un getter et un setter pour cet attribut. Implémenter une autre classe *Fifille* qui hérite de *Maman* et définit un attribut privé entier, un constructeur qui initialise son attribut par héritage de son paramètre et une méthode qui modifie son attribut en utilisant le setter par héritage de la classe *Maman*.

➤ **Transtypage ascendant (upcasting)**

L'héritage permet de définir une classe en tant que nouveau type d'une classe existante. Ces nouveaux types, par définition, ont toutes les méthodes et attributs de la classe existante. N'importe quel message normalement envoyé à un objet de la superclasse peut donc être envoyé à un objet de la sous-classe. Cela veut dire qu'on peut traiter un objet d'une sous-classe comme si c'était un objet de la superclasse.

Upcasting (transtypage ascendant) nous permet de traiter sans distinction toutes les sous-classes d'une même superclasse

Exercice 15 : Implémenter une classe *Personne* qui définit un attribut privé comme un objet conjoint d'une *Personne* et une méthode qui permet de *pacser* une personne à une autre. Implémenter 2 autres classes *Homme* et *Femme* héritant de *Personne*. Dans une dernière classe *Test*, écrire le **main** qui instancie des objets des classes *Homme* et *Femme* et appelle la méthode *pacser* pour chacun de ces objets.