



- ❖ Plan de cours 
- ❖ Database-first vs Code-first 
- ❖ Entités et attributs 
- ❖ Relations



- ❖ Lorsqu'on développe une application qui implique une **base de données**, nous avons le choix entre ces approches :

Code-first

On crée les **Models** (classes) dont on a besoin dans notre application PUIS on peut générer la **base de données** qui s'adaptera à nos classes.

Classes 



BD 

Database-first

On crée une **base de données** (avec des schémas, des tables, ...) PUIS on génère des **Models** dans notre application qui s'adapteront à notre base de données.

BD 



Classes 

- ◆ Dans les deux cas, on peut utiliser un **ORM** (Object-Relational Mapping) comme **Entity Framework** qui va générer la BD (ou les Models) et s'assurera de la **compatibilité** entre les deux.



❖ Bases de données et programmation Web (ce cours)

◆ Database-first

- Tous les cours de **Web** et de **mobile** de votre cheminement collégial sont **Code-first**... sauf celui-ci !
- Difficile de dire si **Code-first** est plus populaire que **Database-first** aujourd'hui, mais l'approche **Code-first** gagne définitivement en popularité pour le développement de **nouveaux projets**.
- Historiquement, **Database-first** était *probablement* plus populaire et il y a actuellement des tonnes de projets (nouveaux ou anciens) **Database-first** à créer / maintenir sur le marché du travail !



❖ Bases de données et programmation Web (ce cours)

◆ Database-first

- Il y a de fortes chances que dans votre travail, vous travaillez avec des projets Database-first!
- Mais comme c'est plus simple, un seul cours dans votre programme sera suffisant pour comprendre comment travailler avec des projets Database-firsts.



❖ Avantages Code-first

- ◆ Possibilité de ne jamais avoir à « créer ou modifier soi-même » la base de données.
- ◆ Meilleure performance pour des opérations complexes au niveau de l'application s'il y a peu d'interactions avec la base de données.
- ◆ *Généralement plus adapté pour une application à **développer rapidement** ou avec des besoins légers au niveau des interactions avec la BD.

❖ Avantages DB-first

- ◆ Maintenance de la base de données : on comprend la base de données (car on l'a créée) et c'est donc plus simple de la documenter et la modifier soi-même.
- ◆ Meilleure performance pour une application qui interagit beaucoup avec la base de données.
- ◆ *Généralement plus adapté pour une application lourde au niveau des interactions avec la BD.

*Il faut être très prudent quand on cherche les avantages et désavantages d'une approche. Cela dépend SURTOUT du type d'application qu'on développe et de notre manière d'utiliser l'approche en question. Si un quidam vous explique que « Code-first c'est toujours mieux » ou que « Database-first c'est toujours mieux », cherchez un peu plus loin car ce n'est pas si simple 



❖ « Stack » technologique du cours

◆ Système de gestion de bases de données : Microsoft SQL Server (MSSQL)

- C'est ce qui était utilisé dans votre cours Introduction aux bases de données. (Donc on ne part pas de zéro) 🧠
- Les concepts que nous aborderons avec MSSQL seront facilement transférables / réutilisables avec d'autres SGBD. (Notamment car SQL est très standardisé) 🔗

◆ Framework Web backend : ASP.NET Core (C#)

- Car vous le connaissez déjà 🧠
- Dans ce cours, on se concentre surtout sur les bases de données et nous n'avons pas vraiment le temps d'apprendre un nouveau Framework ⌚

◆ Framework Web frontend : Aucun 🙋

- En Programmation Web orientée services, vous apprendrez Angular, mais nous ne en servons pas dans ce cours : ASP.NET Core générera le HTML + CSS + JavaScript.



- ❖ Bases de données non relationnelles (« NoSQL »)
 - ◆ Pas abordé dans ce cours. Brièvement abordé en applications mobiles avancées.
 - ◆ C'est quoi ? 🤔 Un type de base de données qui ne stocke pas les données dans des « tables » traditionnelles comme une BD relationnelle.
 - Les données sont plutôt stockées dans ...
 - 📄 **Documents** : Documents JSON, BSON, XML. Cas classique de BD NoSQL. Usage varié.
 - 🔑 **Tables clé-valeur** : (« Tables » avec seulement 2 colonnes : une clé et une valeur) Adapté pour des données simples comme des préférences utilisateur, des profils d'utilisateurs, etc.
 - 📊 **Tables orientée colonnes** : Ces tables sont lues colonne par colonne (plutôt que rangée par rangée) Surtout adapté pour certaines applications analytiques qui font beaucoup d'agrégations de données.
 - ⭐ **Graphes** : Structure qui met l'accent sur les « connexions / lien » entre des données. Usage très particulier. (Ex. réseaux sociaux, réseau de connaissances)
 - ◆ Intérêt des BD NoSQL : Mieux adapté à certains types de projets que les BD relationnelles. (Performance et vitesse de développement)



❖ **Modélisation** (Semaine 1 du cours)

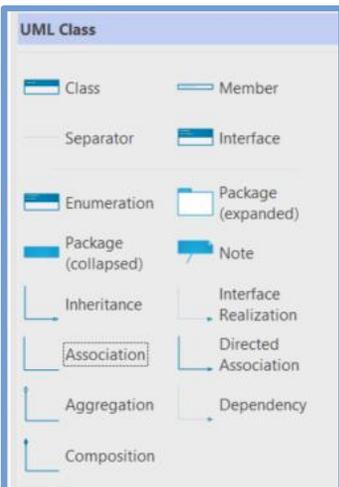
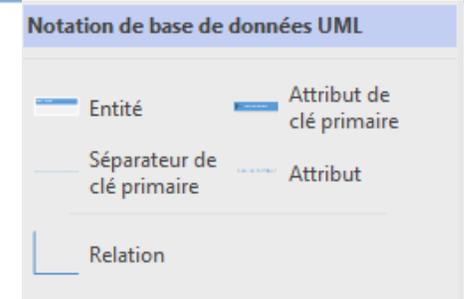
- ◆ **Diagrammes** qui nous permettront de déterminer la **structure** de notre base de données (entités, relations, attributs, etc.)
- ◆ Créer des tables sur le pouce sans d'abord préparer de diagramme ? 
 - Plus d'**erreurs** imprévisibles pendant le développement
 - Mauvaise **maintenabilité** (Difficile de faire des changements plus tard)
 - **Performances** moins optimisées
 - **Redondance** des données
 - Bref, plein de problèmes qui **coûtent cher** à réparer ou endurer ! 
- ◆ C'est pour ça que nous réviserons la **modélisation** et aborderons même de nouvelles notions qui y sont liées.
 - La modélisation peut parfois sembler fastidieuse ou abstraite, mais elle nous rend un énorme service ! 



❖ Modélisation

◆ Modèle conceptuel Unotation de base de données UML

- Premier type de schéma que nous allons aborder 🐧
 - À partir de ce schéma « **conceptuel** », il sera plus simple d'élaborer un diagramme **logique** pour ensuite créer la base de données. On ne crée pas la base de données à partir du modèle entité-association !
- Découle des **besoins du client** 🧡
 - Il faudra lire du texte (communiquer avec un client) pour connaître quelles **données** seront nécessaires au **fonctionnement de l'application** et quelles **relations** existent entre ces différentes données.

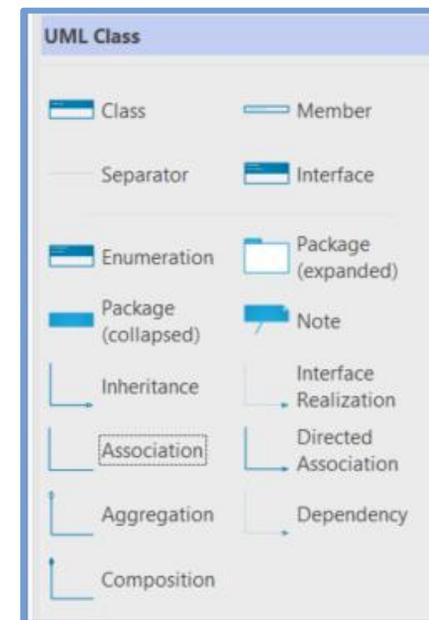
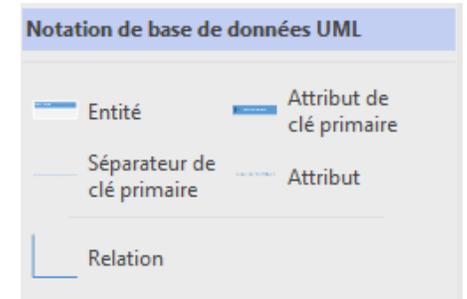


draw.io



❖ Modélisation

- ◆ **Modèle conceptuel** **Notation de base de données UML**
 - Pour créer nos entités et définir les clés primaires et les clés étrangères
- ◆ **UML Class**
 - Pour définir les relations entre les entités.





❖ Modélisation

◆ Des modèles conceptuels variés et mélangeants 🤪

- Il existe de nombreux types de modèles conceptuels. (Qui se ressemblent généralement)
- Pour la plupart des types de modèles conceptuels, certaines **notations** (ex. utiliser un losange pour une relation VS utiliser **une simple ligne pour une relation**) ne font pas l'unanimité : ce n'est pas grave 😊
 - L'important, c'est de comprendre une manière de faire et de s'assurer qu'elle nous aide à avancer dans les prochaines étapes du processus de conception de la base de données.
- Dans tous les cas, si vous cherchez des exemples supplémentaires sur le Web, gardez à l'esprit que les notations utilisées pourraient varier et même parfois se contredire entre elles 😬



❖ Entité

- ◆ Objet ou concept généralement lié à plusieurs caractéristiques. (**attributs**)
 - L'entité est une boîte (class) et les attributs (member) sont à l'intérieur.



◆ Exemple 1

- « On souhaite conserver les données de nos **clients** : leur **prénom**, leur **nom**, leur **adresse**, leur **numéro de téléphone** et leur **adresse courriel**. »

Client
Prénom
Nom
Adresse
Adresse courriel
Numéro de téléphone

Si vous voulez, vous pouvez **raccourcir** ou **simplifier** les noms des **attributs**, tant qu'ils restent très clairs. Au stade du modèle conceptuel, on doit avant tout miser sur la clarté.



❖ Entité

◆ Exemple 2

- « L'application devra permettre de faire la liste de toutes les **maisons**. Il faudra pouvoir afficher le **prix**, le **nombre de pièces**, la **date de construction**, le **nombre de salles de bain** et la **superficie du terrain** lorsqu'on clique sur une maison. »

Maison
Prix
Nombre de Pieces
Adresse (? voir Client)
Date de Construction
Nombre de salles de bain
Superficie du terrain



Le client n'a pas mentionné « Adresse », mais ça semble être une donnée incontournable pour une maison. On se garde une note sous le schéma !

Maison

Adresse : Le client n'a pas parlé de cet attribut. À vérifier avec lui.



❖ Entité

◆ Exemple 3

- « Chaque **joueur** peut accéder à son profil pour modifier certaines informations. On aimerait par exemple qu'il puisse spécifier son **adresse courriel**, son **numéro de carte de crédit**, sa **date de naissance**, une **phrase** ou une **description personnelle**, son **prénom** et son **nom**. »

Joueur
Prénom
Nom
Phrase personnelle
Date de naissance
No de crédit
Adresse courriel

Les mots « **joueur** » et « **profil** » pourraient porter à confusion. Est-ce que c'est joueur ou profil l'**entité** ?

Dans ce cas, on a interprété le **joueur** comme étant l'**entité de données** et le **profil** étant une *fonctionnalité* qui donne accès aux données du **joueur**.



❖ Entité

◆ Exemple 3 (suite)

- « Chaque **joueur** peut accéder à son profil pour modifier certaines informations. On aimerait par exemple qu'il puisse spécifier son **adresse courriel**, son **numéro de carte de crédit**, sa **date de naissance**, une **phrase** ou une **description personnelle**, son **prénom** et son **nom**. »

Joueur
Prénom
Nom
Phrase personnelle
Date de naissance
No de crédit
Adresse courriel

Le client a mentionné une **phrase** ou une **description personnelle**.

Il faut clarifier avec le client ce qu'il veut que l'application utilise car on ne peut pas utiliser les 2.

Dans ce cas, le client nous a demandé d'utiliser **Phrase personnelle**

On se garde une note sous le schéma !

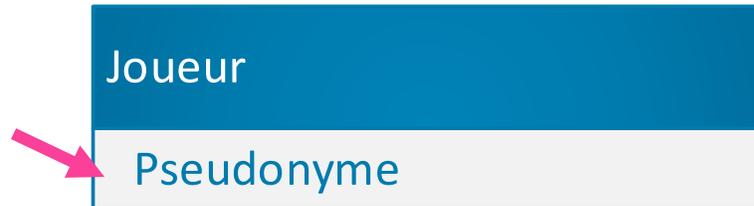
Joueur

Phrase personnelle: synonyme de 'Description personnelle'



❖ Attributs

- ◆ Donnée (caractéristique) liée à une entité.
- ◆ **Attributs atomiques**
 - Contiennent une seule valeur.



- Exemple 1 : Un joueur possède un seul pseudonyme. (Un joueur ne peut pas être simultanément nommé **CampingSniper23** et **HelloKitty04**)



- Exemple 2 : Un compte chèques possède un seul solde. (Un compte chèques ne peut pas simultanément avoir un solde de **23.04\$** et **364 210.78\$**)



❖ Attributs

◆ Attributs à valeurs multiples

- Peuvent avoir plus d'une valeur. Identifiés par un domaine de valeurs [X..Y]

- Exemple : On souhaite pouvoir noter, pour un même client, un à trois **numéros de téléphone**, plusieurs **adresses** et plusieurs **adresses courriel**.

(Le client pourrait être associé au numéro de téléphone **514-819-2244** et **514-420-6969**)

Client
Prénom
Nom
Adresse [1..N]
Courriel [1..N]
No de téléphone [1..3]

On voit qu'un client peut avoir 1 à « infini » adresses et courriels, ainsi que 1 à 3 numéros de téléphone.



❖ Attributs

◆ Attributs optionnels ? 🔍

- Peuvent être « null ». (Vides) Identifiés par le domaine de valeurs [0..X]

• Exemple : Lorsqu'on ajoute un client dans la base de données, il n'est pas obligé de fournir son adresse courriel. 📧 😞

Client
Prénom
Nom
Adresse
Adresse courriel [0..1] 
Numéro de téléphone

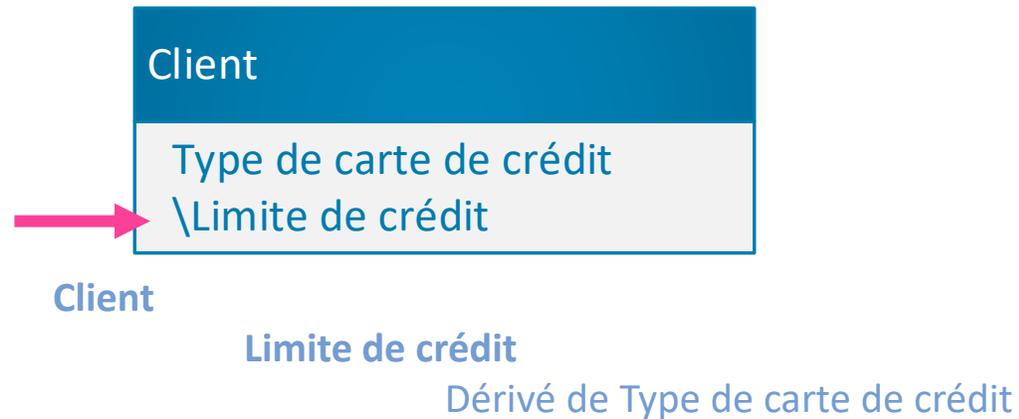


❖ Attributs

◆ Attributs dérivés 🧐

- Leur valeur est dérivée d'un autre attribut. Identifié avec \

Exemple 2 : Le **type de carte de crédit** détenu par un client bancaire détermine sa **limite de crédit** disponible (Un client avec une carte de crédit **Ultra Deluxe Gold Bingo Bango Fiesta** dispose d'une limite de crédit de **20 000\$**)



On **doit** ajouter une précision près de la table pour garder à l'esprit de quel autre attribut c'est dérivé 🧐



❖ Attributs

◆ Attributs dérivés 🧒

- Leur valeur est dérivée d'un autre attribut. Identifié avec \



Exemple 1 : La valeur de l'âge est déterminée à partir de la date de naissance.
(On calcule $\text{âge} = \text{aujourd'hui} - \text{date de naissance}$ au lieu de demander à l'utilisateur son âge !)

Cet exemple est un classique, mais c'est une **mauvaise pratique**.

Créer un champ pour l'âge dans une base de données n'est pas stratégique car **c'est une valeur qui change tout le temps et c'est une donnée qui peut être obtenue dans une requête, au moment où on en a besoin...**

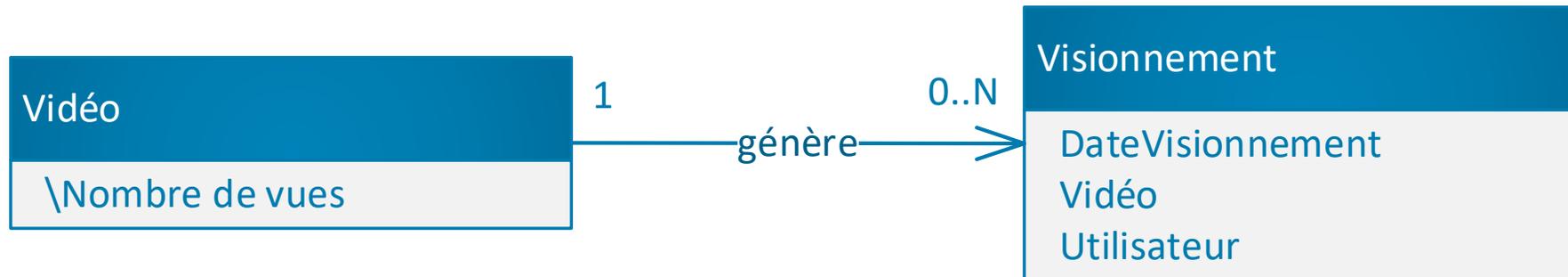
Imaginez une base de données avec 10 millions d'utilisateurs : Si on fait un champ calculé, chaque requête sur la table Utilisateur calculera l'âge de l'utilisateur, même si dans la requête on n'a pas besoin de l'âge!



❖ Attributs

◆ Attributs dérivés 🧐

- Leur valeur est dérivée d'un autre attribut. Identifié avec \



Parfois un attribut est dérivé d'une **donnée située dans une autre entité**. Par exemple, une vidéo comporte un certain nombre de **vues** (Ex. une vidéo a été regardé **36 400** fois) et pour chaque visionnement de vidéo, on note la **date**, la **vidéo visionnée** et l'**utilisateur** qui a visionné.

À chaque fois qu'un **visionnement** de la vidéo « Tongue sounds ASMR » est ajouté, l'attribut **Nombre de vues** de la vidéo « Tongue sounds ASMR » doit augmenter de 1. C'est quelque chose qui est mis-à-jour grâce à un déclencheur de type INSERT sur la table Visionnement.

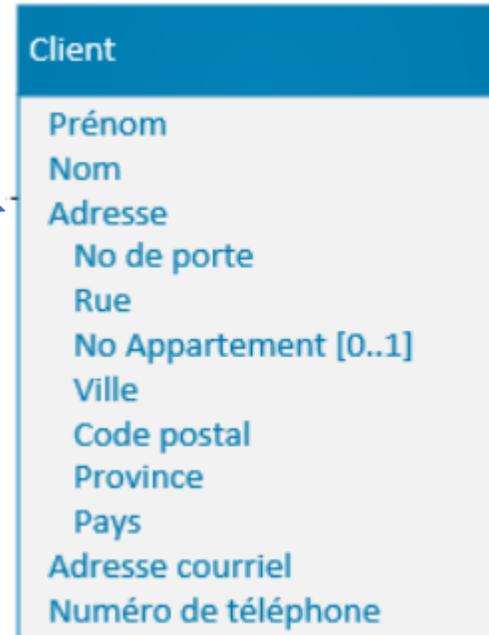


❖ Attributs

◆ Attributs composites 🐙

- Contiennent plusieurs autres attributs. Identifié ici en mettant en retrait les sous-attributs.

Exemple : Une **adresse** est représentée par plusieurs valeurs également.





❖ Attributs

- ◆ **Attributs composites** : Faut-il absolument préciser toutes les composantes d'un attribut composite ? 😞 😬
 - **Oui** : Si les besoins exprimés par le client **indiquent clairement** de quelles sous-informations est composée une adresse. À ce moment, c'est important d'ajouter le plus de détails possibles dans le modèle pour créer une base de données offrant plus de flexibilité ensuite.
 - **Non** : Si les besoins exprimés par le client mentionnent seulement « adresse ». Une simple **chaîne de caractères** pourrait suffire à représenter la donnée. (À confirmer avec le client)*

*Attention au **format** des données composites ! Si on représente l'adresse avec un simple **string** (varchar) dans la base de données, (ex. "54 rue des dromadaires, Chambly, QC") manipuler les adresses lors des requêtes SQL pourrait devenir moins simple.

Si on sait d'avance que certaines fonctionnalités vont exploiter les adresses, il vaut mieux élaborer un format de donnée plus précis.

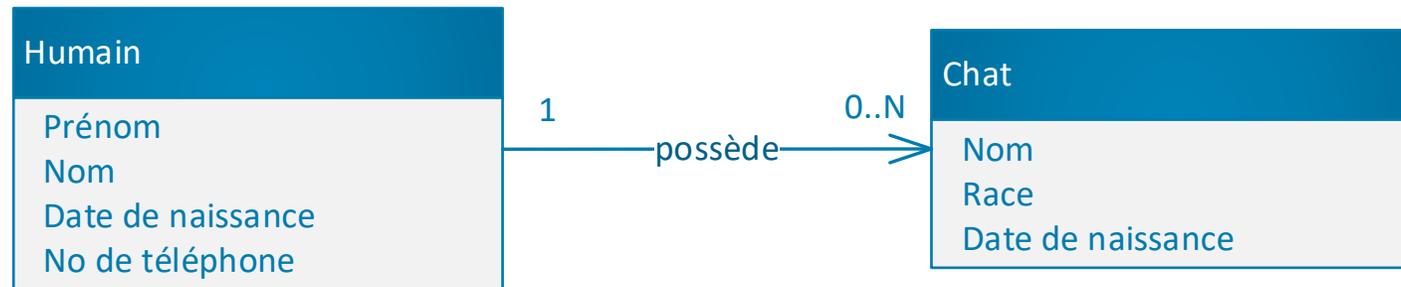
Client
Prénom
Nom
Adresse
Adresse courriel
Numéro de téléphone

Client
Prénom
Nom
Adresse
No de porte
Rue
No Appartement [0..1]
Ville
Code postal
Province
Pays
Adresse courriel
Numéro de téléphone



❖ Relations 🗨️ 👤

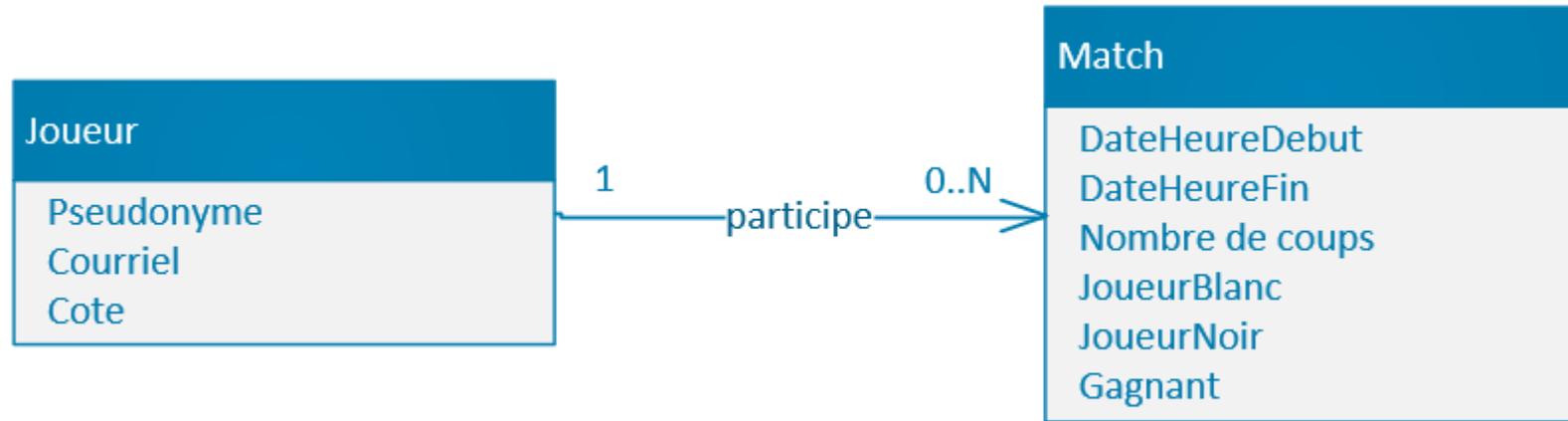
- ◆ Permettent de faire le lien entre des entités. (Donc entre des données)
- ◆ Représentées par des **flèches**.
- ◆ *Généralement* nommées avec un **verbe** à l'indicatif présent.



Exemple 1 : « Un **humain** **peut posséder** un **chat**. Dans la base de données, on souhaite noter le **nom**, la **date de naissance** et le **numéro de téléphone** de chaque humain et on souhaite noter le **nom**, la **race** et la **date de naissance** de chaque chat. »



❖ Relations



Exemple 2 : « Un **match d'échecs** se déroule entre deux **joueurs**, un qui a les pièces blanches et un qui a les pièces noires, qui débute à une date et une heure précise. Dans le système, on souhaiterait noter le **nombre de coups joués**, la date et l'heure de la fin de la partie et le **gagnant**. On aimerait aussi noter le pseudonyme, la **cote** et l'**adresse courriel** de chaque participant à un match. »

On remarque que la description des besoins du client est plus vague qu'à l'**exemple 1** ! Il faut réfléchir et interpréter :

- Pour chaque attribut, on doit se demander si la valeur est liée à un **joueur** ou à une **partie d'échecs**.
- La relation entre un **joueur** et un **match** n'est pas mentionnée explicitement, mais il faut la définir.



❖ Relations



Exemple 3 : « Nos **clients**, identifiés par leur **nom complet** et leur **numéro de téléphone**, peuvent nous acheter des **voitures**. Évidemment, une voiture correspond à un **modèle**, une **année**, un **type de moteur** (hybride, électrique ou essence), un **prix** et un **fabricant**. Pour le fabricant, on aura les informations suivantes: **Site Web** du fabricant, **nom du fabricant** et **numéro de téléphone**.»

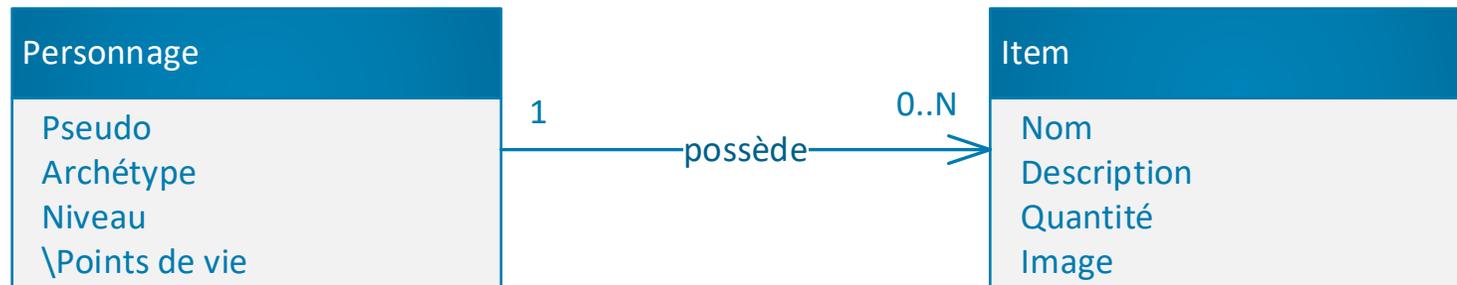
Cette fois-ci, le **fabricant** pourrait facilement être interprété comme un **attribut composite**. Étant donné qu'un fabricant **produit** probablement **plusieurs voitures** **ET** qu'on a des **attributs sur le fabricant**, il est préférable d'en faire une **entité** !



❖ Relations

◆ Cardinalité (ou « multiplicité »)

- La cardinalité exprime le nombre d'instances d'une entité qui peut être associé à l'instance d'une autre entité.
 - « **1** individu est marié avec **1** autre individu. » (One-To-One)
 - « **1** client peut acheter **N** voitures. » (One-To-Many)
 - « **N** joueurs peuvent être amis avec **M** joueurs. » (Many-To-Many)
- Une cardinalité est indiquée à l'aide de nombres sur le trait d'une relation.
- Exemple 1 : « Un **personnage** peut posséder plusieurs **items** dans son inventaire. »



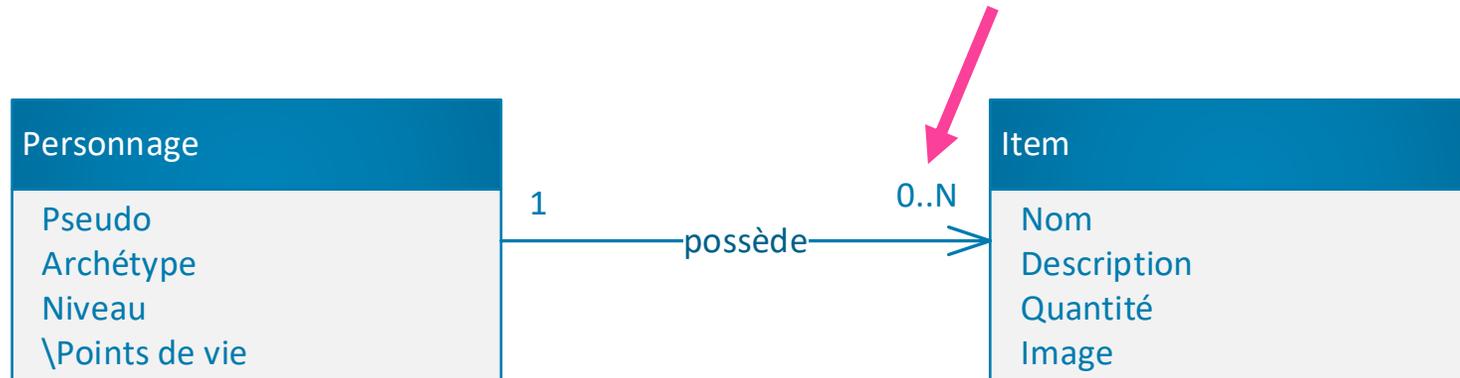


❖ Relations

◆ Cardinalité

- Exemple 1 : « Un **personnage** peut posséder plusieurs **items** dans son inventaire. »

Chaque personnage peut* posséder **plusieurs items**. (donc de 0 à N ici)



~~Chaque item est détenu par un personnage.~~ (donc 1 ici)

*S'il n'y avait pas eu « peut » dans la phrase, on aurait pu mettre 1..N et interpréter qu'un personnage possède au moins 1 item.



❖ Relations

◆ Cardinalité

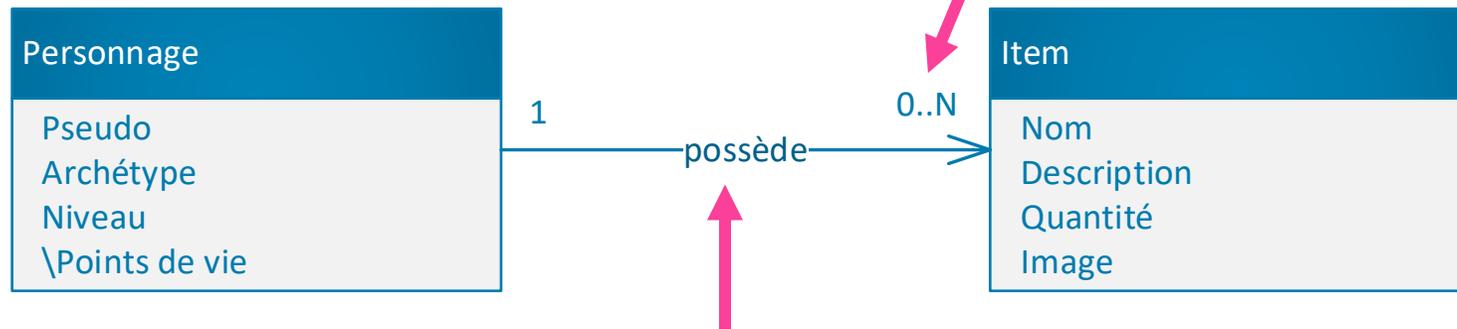
- Exemple 1 : « Un *personnage* peut posséder plusieurs *items* dans son inventaire. »

Chaque personnage peut* posséder **plusieurs items**. (donc de 0 à N ici).

PHRASE ACTIVE

Chaque item **est détenu** par **un personnage**. (donc 1 ici)

PHRASE PASSIVE



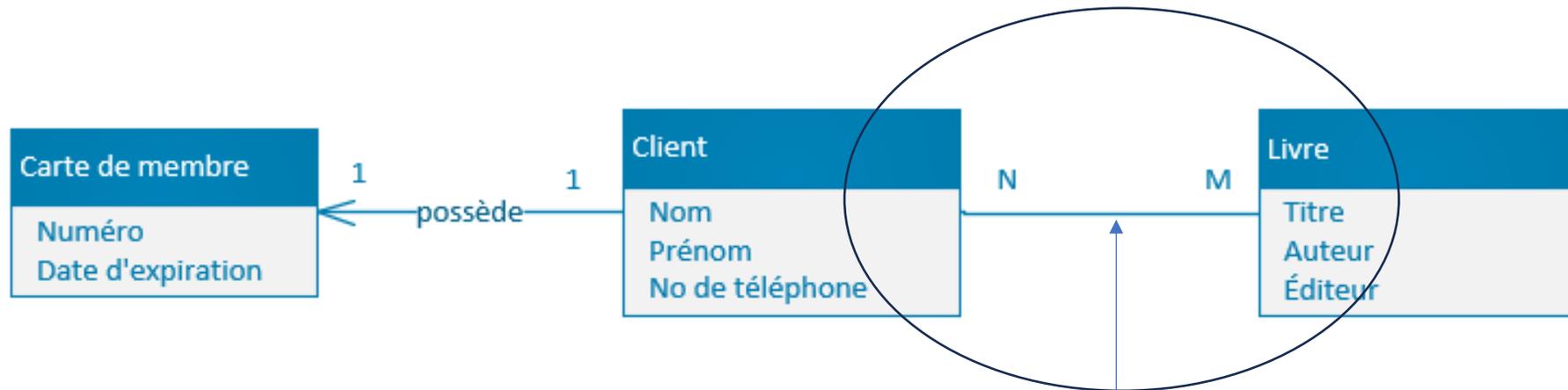
On garde la flèche dans la direction ACTIVE



❖ Relations

◆ Cardinalité

- Exemple 2 : « Dans une bibliothèque, chaque **client**, nom complet et numéro de téléphone, possède une **carte de membre**, Date d'expiration et numéro de carte. De plus, chaque **client** peut louer des **livres** »



Relation plusieurs à plusieurs
notée N-M, sans flèche ni verbe d'action



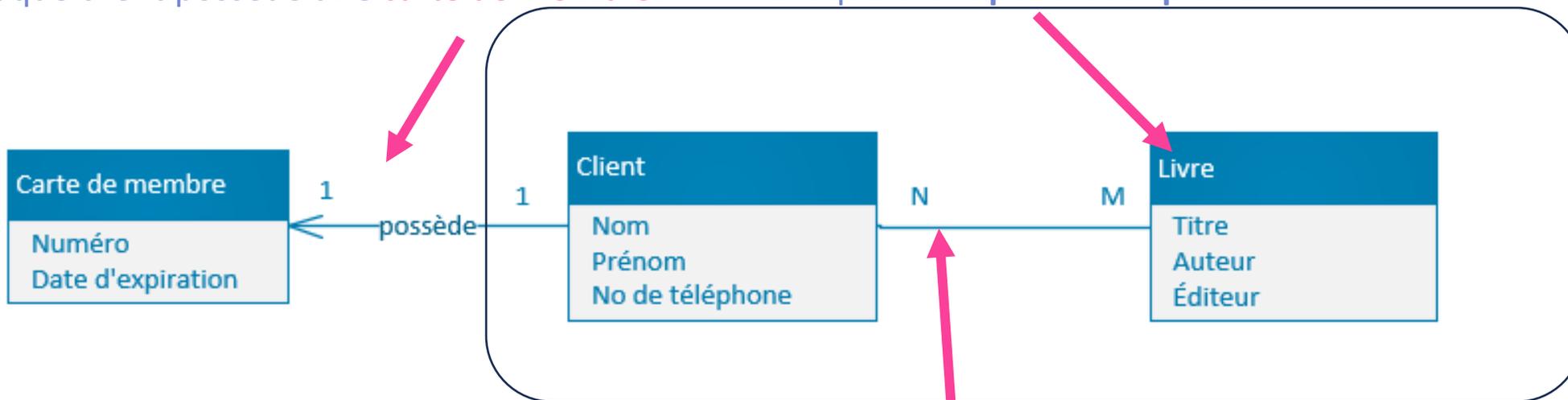
❖ Relations

◆ Cardinalité

- Exemple 2

Chaque client possède **une** carte de membre.

Chaque client **peut** louer **plusieurs** livres



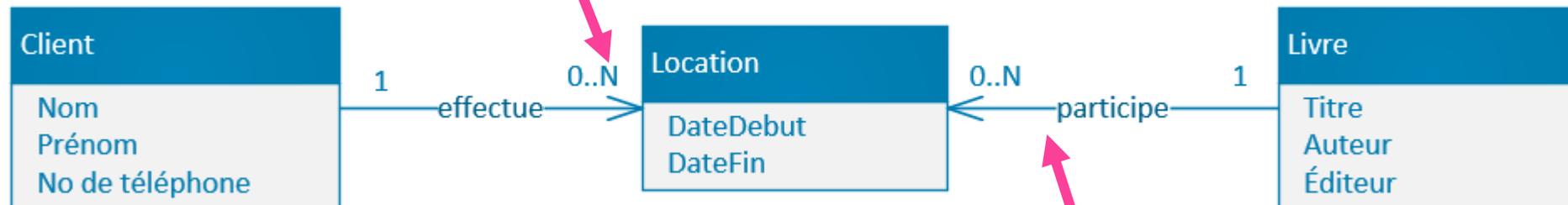
Chaque livre **peut** être loué par **plusieurs** clients



❖ Relations

- ◆ Briser la relation plusieurs à plusieurs avec une table associative quand le cas mentionne l'entité intermédiaire.
 - Exemple 3 : « Dans une bibliothèque, chaque **client** (nom, prénom et numéro de téléphone) peut louer des **livres**. Lorsqu'une location est terminée, un livre peut être loué par un autre client »

Chaque client **peut effectuer plusieurs** locations.

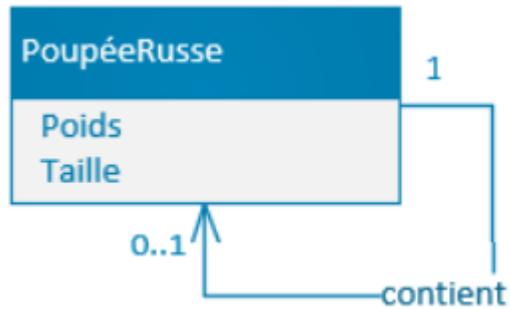


Chaque livre **peut être loué plusieurs** fois

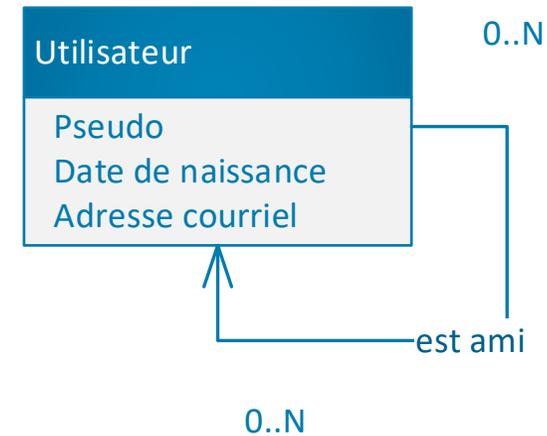


❖ Relations

- ◆ Relations réflexives 🐱 (Entité en relation avec elle-même)



Exemple 1 : « Une **poupée russe**, dont on note le poids et la taille, peut contenir **une** autre **poupée russe**. (Et être contenue dans une autre poupée russe) »



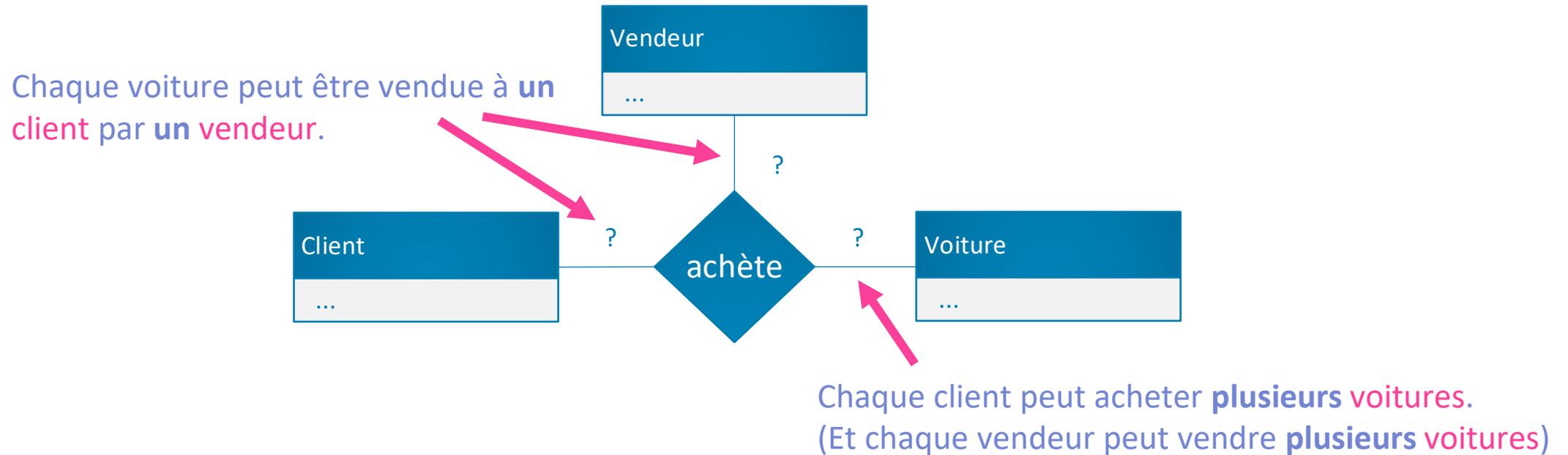
Exemple 2 : « Un **utilisateur** (représenté par un pseudo, un courriel et une date de naissance) **peut** être ami avec **plusieurs** autres **utilisateurs**. »



❖ Relations

◆ Relations avec un **degré*** supérieur à 2 🤪

- Exemple : « Un **client** peut acheter des **voitures**. Lors d'un achat, un **vendeur** spécifique est impliqué dans la transaction étant donné qu'il travaille à la commission. »



*Le **degré** d'une relation correspond au nombre d'entités différentes impliquées dans la relation



❖ Relations

◆ Relations avec un degré supérieur à 2

- Les relations de degré supérieur à 2 sont généralement transformées par l'ajout d'une entité
- Nous avons pris la même situation, mais nous avons décidé d'ajouter l'entité « **Achat** » pour faire le pont entre les trois autres entités :

