Conception UML et Model Driven Development

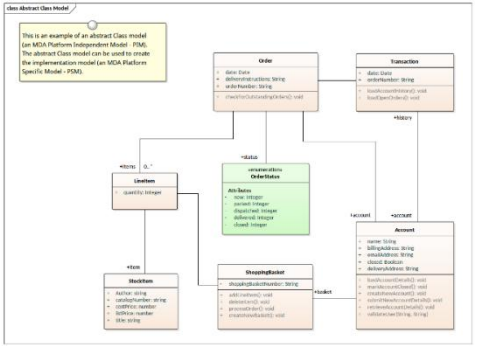
Programme détaillé ou sommaire

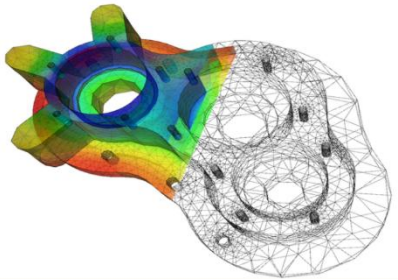
Introduction
UML
Rappels - Diagramme de cas d’utilisation
Diagramme d'activités
Rappels - Diagramme de classes
Diagramme entités relations
Bonnes pratiques de conception
Diagramme de séquences
Introduction au Model Driven Development

Introduction

Rappels – Définition modèle

Un **modèle** est une **représentation abstraite** et **simplifiée** (exclut certains détails) d'une entité du monde réel (système, phénomène, processus, etc.) en vue de le décrire, de l'expliquer ou de prédire.





Pourquoi modéliser ?

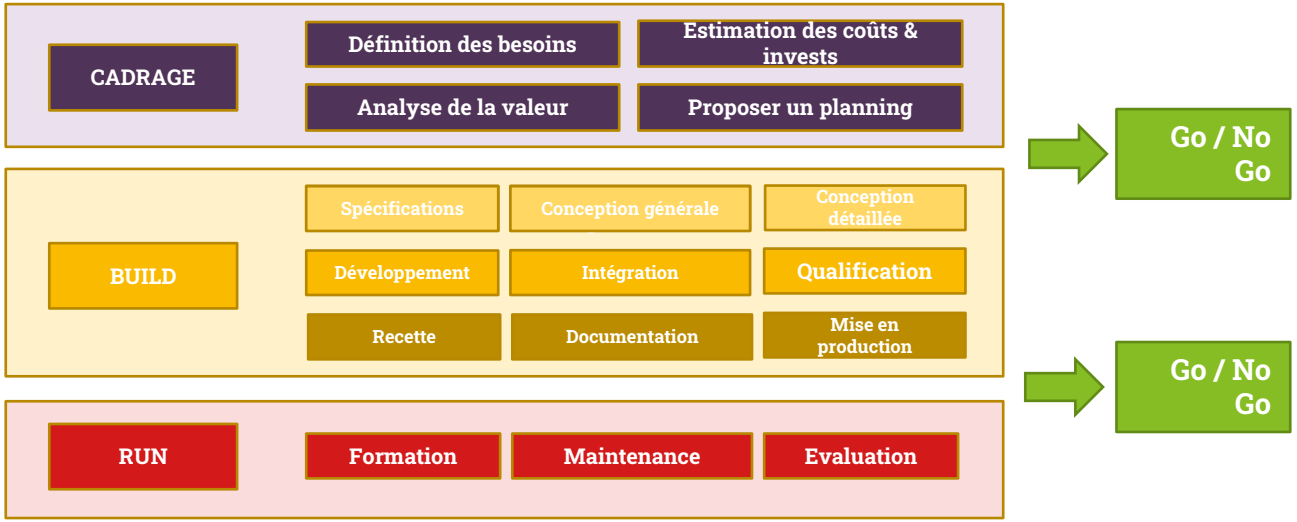
- **un petit dessin vaut mieux qu'un long discours.**

- Un **modèle** est une abstraction permettant de mieux comprendre un objet complexe (bâtiment, économie, atmosphère, cellule, logiciel, …).

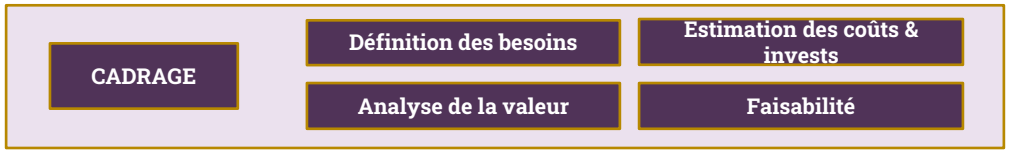
- Un **modèle** sert :
  - de document d'échange,
  - d'outil de conception,
  - de référence pour la réalisation,
  - de référence pour la maintenance et l'évolution

- La construction d'un système d'information, d'un réseau, d'un logiciel complexe, de taille importante nécessite une **modélisation**.

Les phases « macro » d'un projet



Le cadrage



Permet de définir le QUOI

Responsabilité MOE (équipe de développement):

- ❑ Bien comprendre les enjeux et les besoins
- ❑ Définir les priorités
- ❑ Etudier la faisabilité technique
- ❑ Estimer les coûts et proposer un planning prévisionnel

Responsabilité MOA (client):

- ❑ Faire l'analyse de la valeur (estimer les gains pour l'entreprise)
- ❑ Confronter le coût vs les gains attendus

→ **Go / No Go**

Le build - les spécifications

Spécifier les besoins avec la MOA et le concepteur:

- ❑ Identifier les fonctionnalités (features) et les cas d'utilisation (use cases).
- ❑ Réaliser un diagramme de cas d'utilisation et éventuellement un diagramme d'activités (enchainement des cas d'utilisation)
- ❑ Maquetter l'application afin que le client puisse se projeter
- ❑ Recenser l'ensemble des règles métier, qui serviront par la suite à l'équipe de tests
- ❑ Exemple de règles métier pour un formulaire de création d'un compte client :
  - Le nom est obligatoire
  - Le prénom est obligatoire
  - L'adresse mail est obligatoire.
- ❑ **Livrable**: Réaliser un dossier exhaustif avec le diagramme de cas d'utilisation et, pour chaque cas d'utilisation, les maquettes et l'ensemble des règles métier.

Le build – la conception générale

Décrire l'architecture cible générale:

- ❑ Identification des échanges avec des applications externes: quelles données ?
- ❑ Définir la stratégie de sécurité
- ❑ Identification des acteurs (dont applications externes) et cas d'utilisation
- ❑ Identification des risques potentiels et des contraintes majeures
- ❑ Esquisse de l'architecture générale sans entrer dans le détail.
- ❑ Définir les contraintes de fonctionnement: temps de réponse, nombre d'utilisateurs, etc.
- ❑ Définir la stratégie de tests utilisateur
- ❑ Définir la stratégie de tests technique (performance) avec les intégrateurs

Le build – la conception détaillée

Décrire l'architecture cible générale:

- Echange avec des applications externes: format des messages
- **Modélisation** : diagramme de classes, diagramme de séquences, modèle physique de données
- **Pour une API** : définir les endpoints : nom, données en entrées, verbes HTTP et réponses HTTP (code HTTP et données en sortie)
- Stack technique (couche logicielle)

Le build – la conception détaillée

Description détaillée de l'architecture:

- ❑ Besoin réseau: fréquence et volume des messages entre applications
- ❑ Scalabilité (capacité à étendre sa consommation en CPU, disque pour accompagner un accroissement du nombre d'utilisateurs)
- ❑ Définir les contraintes d'exploitabilité et de maintenabilité (compatibilité avec les outils de monitoring de l'entreprise, facilité de déploiement, d'arrêt/relance, stratégie de logs, etc..)
- ❑ Dossier d'intégration (consignes pour installer l'application sur un serveur)
- ❑ Dossier d'exploitabilité de l'application: Consignes d'A/R de l'application. Peut-on relancer un traitement qui a échoué ? PRA (Plan de Reprise d'Activités) ? Procédures pour un retour-arrière ?

UML

Outils de modélisation

- **Langage de modélisation** : une syntaxe commune, graphique, pour modéliser (UML, MERISE,…).

- **UML**: le langage standard pour la modélisation objet.

- Les spécifications UML font environ 800 pages et couvrent tous les aspects du développement logiciel, offrant des outils de modélisation de l'architecture générale aux plus petits détails.

Que peut-on faire avec UML ?

- **UML** permet de modéliser une application :
  - Définir les acteurs humains, ou non, qui vont interagir avec l'application.
  - Définir les fonctionnalités attendues.
  - Définir le contexte technique (serveurs, réseau, etc...)
  - Définir le fonctionnement dynamique de chaque fonctionnalité.
  - Décrire les divers éléments du système, leur cycle de vie respectif et leurs interactions

- **UML** possède sa propre grammaire graphique : **diagramme**.
- **UML** ne propose aucune démarche projet, aucun processus. Il s'agit uniquement de notations graphiques pour modéliser.

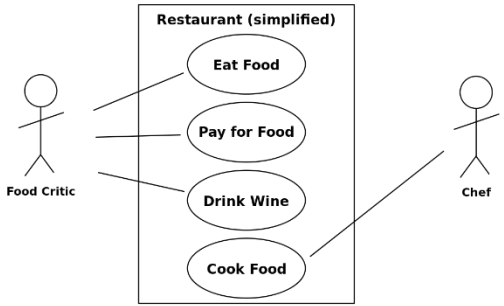
UML et développement informatique

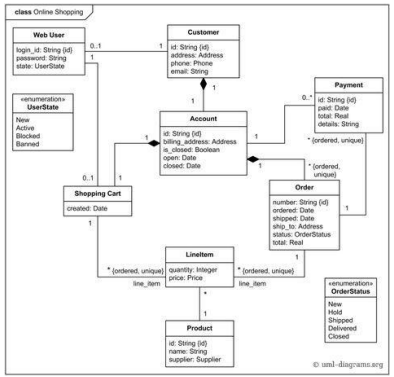
- La plupart des grands IDE (environnements de développement) proposent la génération de code à partir de diagrammes UML.
  - Squelettes de classes
  - Squelettes de méthodes
  - Relations d'héritage

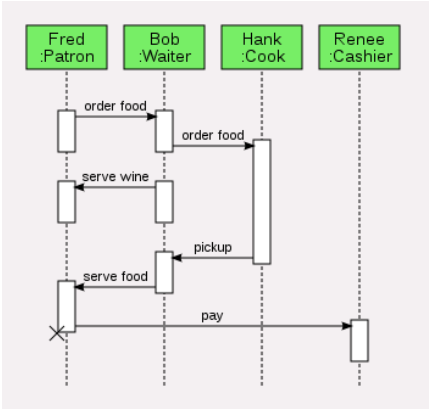
- C'est d'ailleurs préconisé par le **MDD** (Model Driven Development)

Les diagrammes UML

- Il en existe **14 diagrammes** différents
  - Les diagrammes structurels
  - Les diagrammes comportementaux







UML durant les phases d'un projet

- **UML** va être utilisé principalement durant
  - **Les spécifications fonctionnelles**
    - Description des besoins fonctionnels (que peut-on faire ? Qui peut faire quoi ?)
    - Description des objets métiers et de leur cycle de vie
  - **La conception**
    - Description de la solution technique (comment on le fait ?)
    - Description des objets applicatifs
    - Description des échanges entre objets

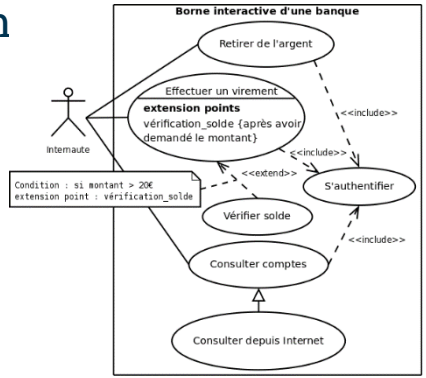
Les spécifications fonctionnelles

Spécifier les besoins avec la MOA et le concepteur

- ❑ Identifier les différents cas d'utilisation (use cases).
- ❑ Réaliser un diagramme de cas d'utilisation et éventuellement un diagramme d'activités (pour modéliser un processus complexe dans le système et des enchainements de cas d'utilisation par exemple)
- ❑ Maquetter l'application afin que le client puisse se projeter
- ❑ Recenser l'ensemble des règles métier, qui serviront par la suite à l'équipe de tests à scénariser les campagnes de tests.

UML durant les spécifications fonctionnelles

- Identification des fonctionnalités et acteurs du système
  - Diagramme de cas d'utilisation



- Modélisation d'un processus pour un cas d'utilisation donné
  - Diagramme d'activités

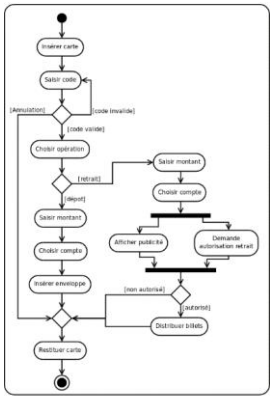
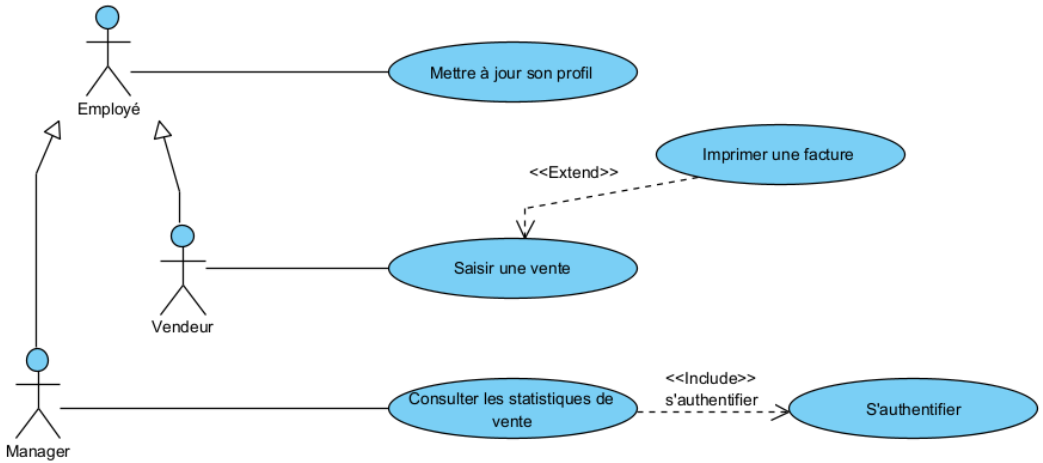


Diagramme de cas d’utilisation

Diagramme de cas d'utilisation

- Il fait partie des diagrammes de comportement (Behavioral diagrams)
- Il décrit les acteurs (qui ?), les fonctionnalités du système (quoi ?) et les interactions entre eux (qui a le droit de faire quoi ?)



Savoir lire un cahier des charges

- Le diagramme de cas d'utilisation est réalisé à partir du cahier des charges.

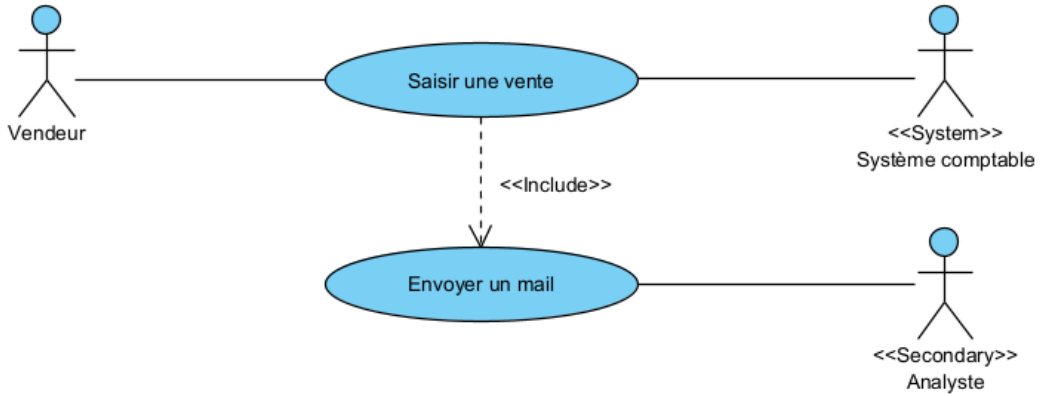
- Savoir lire un cahier des charges émis par le client
  - Dans un cahier des charges, il faut identifier les « **cas d'utilisation** », ou use cases
    - Exemple: L'administrateur saisit une nouvelle référence de livre dans le système
  - Un **cas d'utilisation** permet de définir
    - Acteurs => sujets
    - Méthodes => verbes
    - Entités métier, ou classes => noms communs

Contexte du système

- Identification des acteurs, i.e. agents externes, en interaction avec le système
  - Acteurs humains affichés sous forme de **stickman** (homme bâton)
  - Les acteurs non humains sont représentés sous forme de **rectangle** (autre système) ou d'un **homme bâton doté d'un stéréotype**.

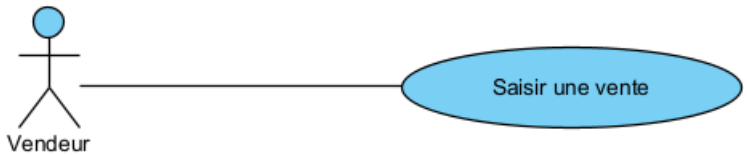
Définir les liens entre acteurs

- Il existe des acteurs principaux et secondaires
  - Les acteurs principaux sont les initiateurs des interactions avec le système
  - Les acteurs secondaires ne sont que des participants.
  - Exemple: un analyste qui reçoit un mail à chaque fois qu'une vente est saisie.



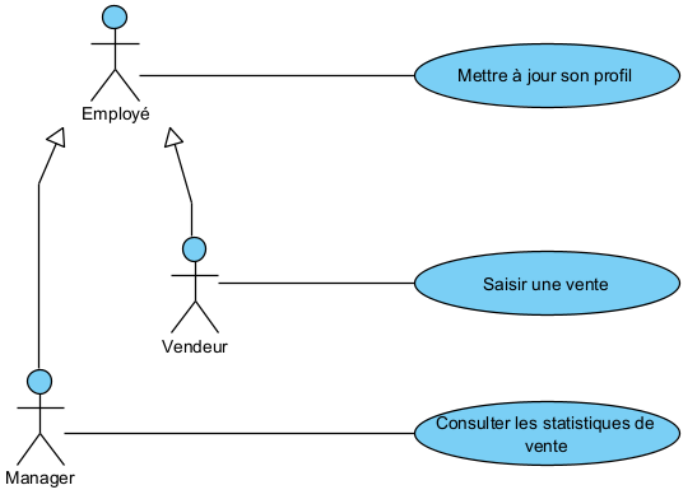
Définir les cas d'utilisation

- Le cas d'utilisation est un élément qui permet de décrire le système
- Le cas d'utilisation représente l'utilisation du système par un acteur.
- Un cas d'utilisation :
  - Comporte un acteur, initiateur du CU
  - Une ellipse contient une description succincte avec un verbe à l'infinitif et un complément.
  - Un lien entre acteur et l'ellipse.



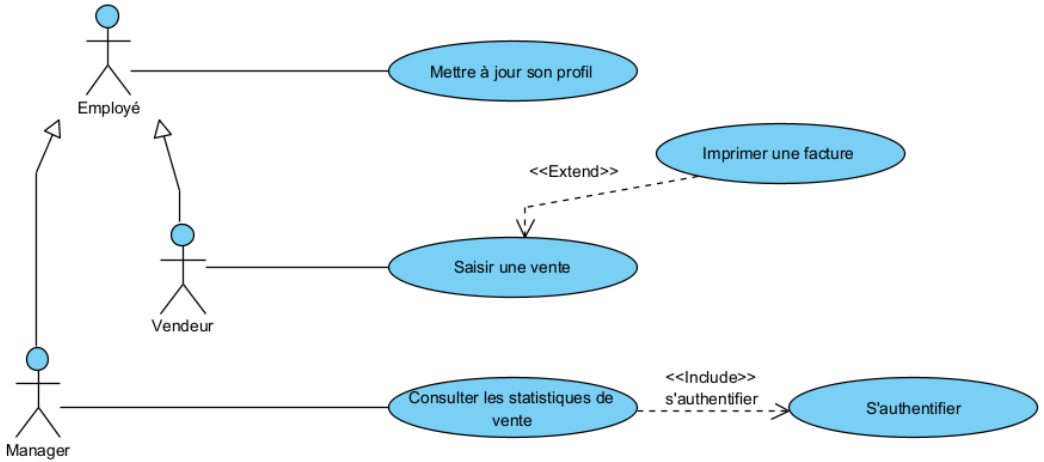
Notions avancées sur les acteurs

- Des acteurs peuvent être reliés par un lien dit de **généralisation**.
- Ce lien est formalisé par une flèche à tête triangulaire.
- Cela permet de factoriser des cas d'utilisation communs.
- Exemple:
  - Le manager et le vendeur sont des employés particuliers.
  - L'un comme l'autre peuvent mettre à jour leur profil



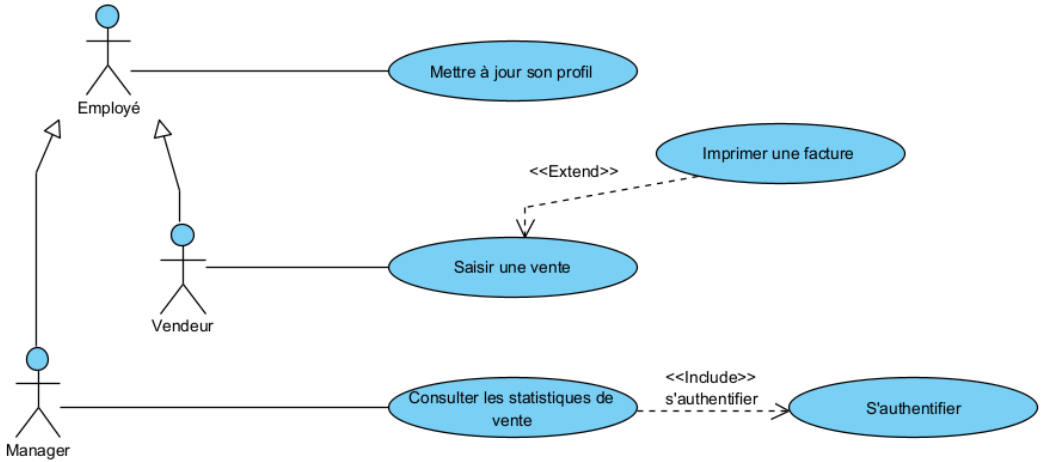
Notions avancées sur les CU

- Comment indiquer qu'un CU possède une ou plusieurs étapes obligatoires ?
- On utilise pour cela un lien de type **include**.
- Exemple:
  - Tous les cas d'utilisation nécessitent une authentification préalable.



Notions avancées sur les CU

- Comment indiquer qu'un CU possède des étapes optionnelles ?
- On utilise pour cela un lien de type **extend**.
- Exemple:
  - Le CU saisir une vente contient une **option** d'impression de la vente au format PDF.



TP n°1

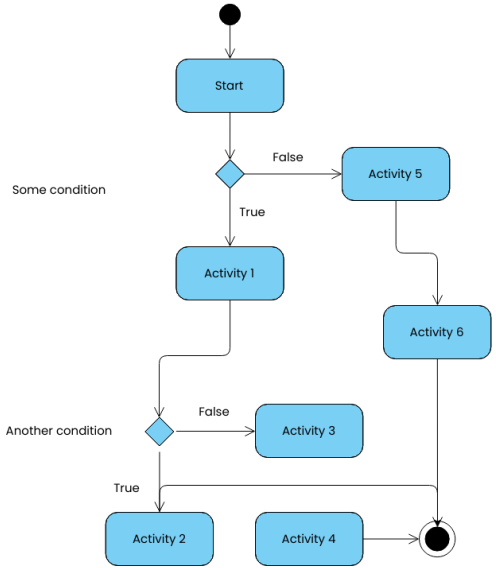
Réalisation d'un diagramme de cas d'utilisation

TP n°1: installez Visual Paradigm puis réalisez le diagramme de cas d'utilisation d'un logiciel de gestion d'un établissement scolaire.

Diagramme d'activités

Diagramme d'activités

- Il fait partie des diagrammes de comportement (Behavioral diagrams)
- Il décrit un processus qui se produit dans le système, en général associé à un cas d'utilisation donné.



Les symboles

- Les symboles les plus courants:
  - Début de l'activité
  - Fin de l'activité
  - Activité
  - Raccord
  - Décision
  - Remarque/Commentaire

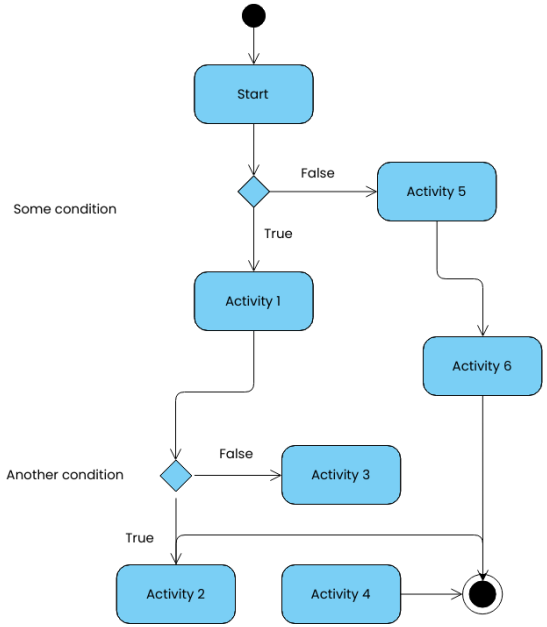


Diagramme d'activités

- Les symboles additionnels:

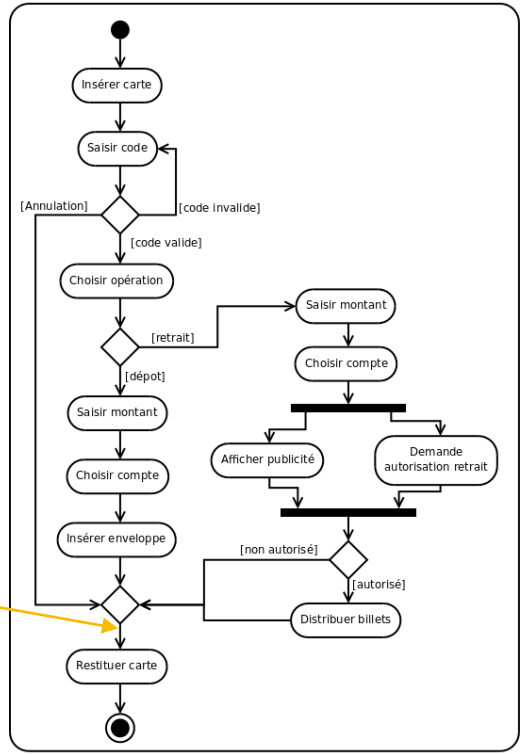
Embranchement
dans le cas d'une // d'activités

Fusion, synchronisation de fin d'embranchement.

Répétition

Exemple

- Distributeur automatique de billets :



A noter que le symbole de décision peut servir de "raccord" et n'avoir qu'une seule sortie

TP n°2

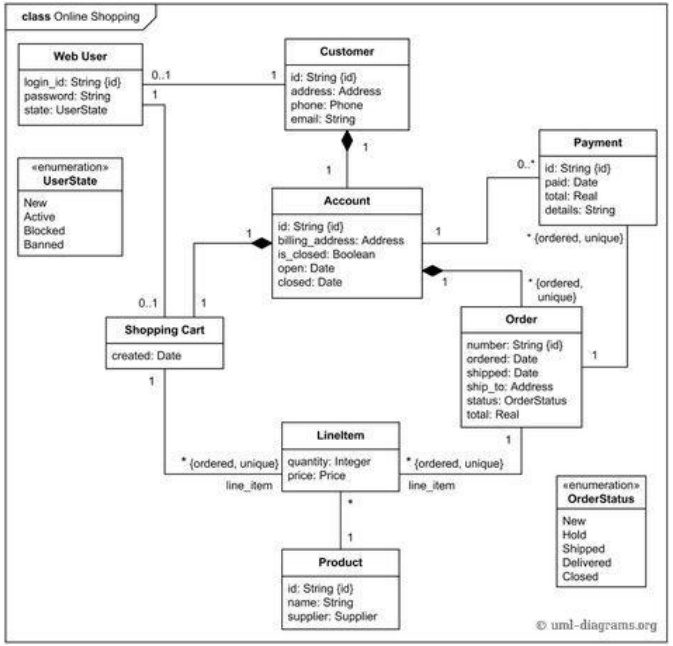
Réalisation d'un diagramme d'activités

TP n°2: réaliser le diagramme d'activités de la page de connexion au système.

Diagramme de classes

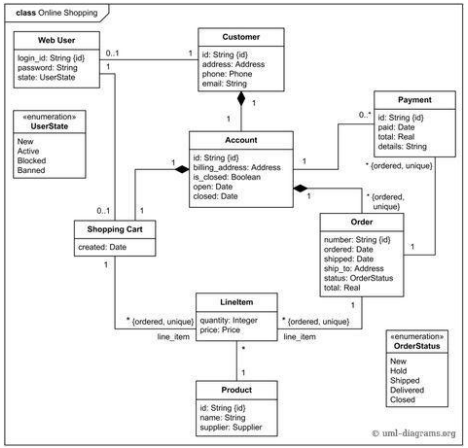
Le diagramme de classes

- Permet de décrire les **différents éléments** du système, ceux pour lesquels on aura **besoin d'une base de données**



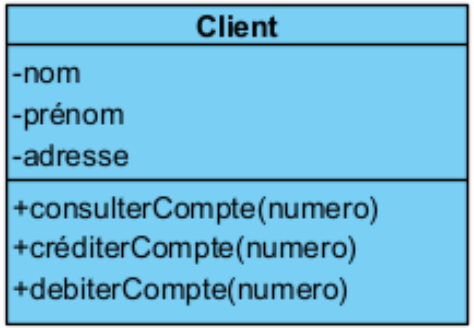
A quoi sert-il ?

- Le **diagramme de classes** sert à décrire la structure d'un système en termes d'éléments et de relations. C'est un diagramme statique, c'est-à-dire que les cycles de vie des objets ne sont pas représentés.
- Il est issu de l'analyse du cahier des charges émis par le client.
- Ce diagramme est présent dans les spécifications fonctionnelles.



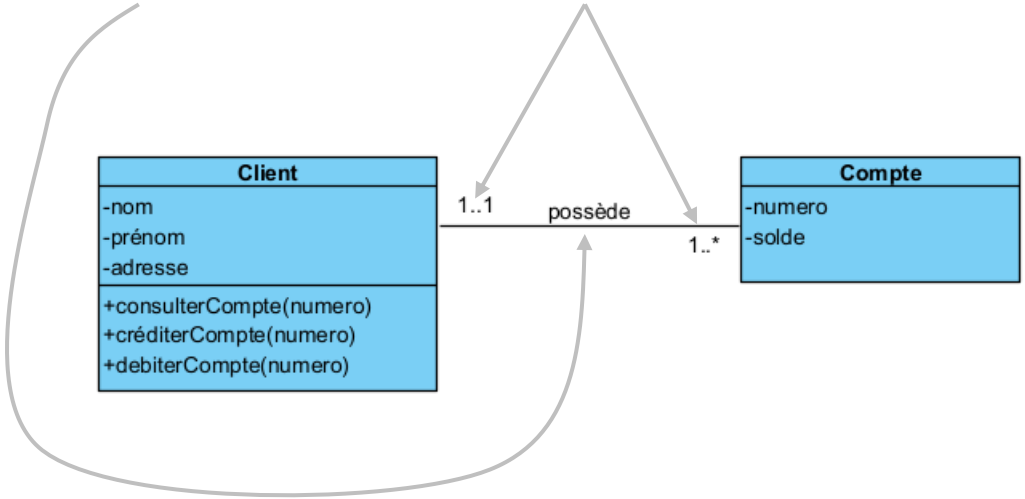
Concept de classe et d'association

- **Classe**
  - Abstraction qui décrit un concept, ses méthodes et ses propriétés
  - Classe = Nom + propriétés + méthodes
- **Propriétés et Méthodes**
  - Une **propriété**, ou attribut, correspond à une information concernant la classe.
  - Une **méthode** correspond à ce que peut faire la classe.
- **Exemple**
  - **Client:** représentation simple et abstraite du client d'une banque.
  - **Propriétés**: nom, prénom, adresse
  - **Méthodes**: consulter, créditer et débiter son compte



Concept de classe et d'association

- **Association** entre classes
  - Une association exprime une **relation**, un **lien** entre deux classes.
  - Cela peut être un lien de possession, de dépendance, etc.
  - On peut y définir le **rôle** de la relation et des **cardinalités**



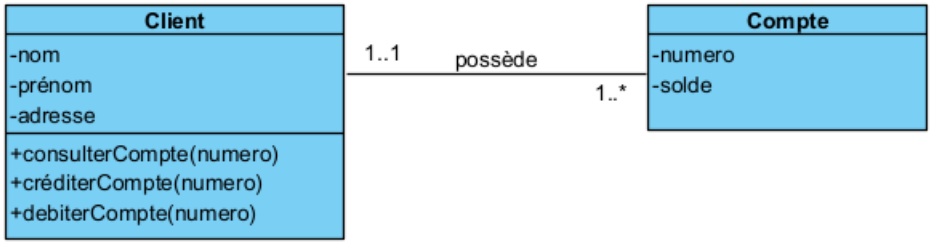
Cardinalité

❑ Définition

- Exprime la multiplicité dans une relation avec un **minimum** et un **maximum**

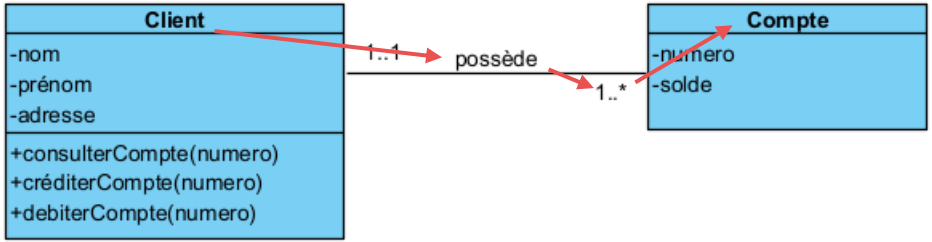
❑ Exemple

- Un client **possède** au moins 1 compte et peut en avoir plusieurs (mais on ne sait pas combien).
- Un compte appartient forcément à 1 client et au plus à un client.
- Le mot « **plusieurs** » est formalisé par une étoile ***** ou **n**

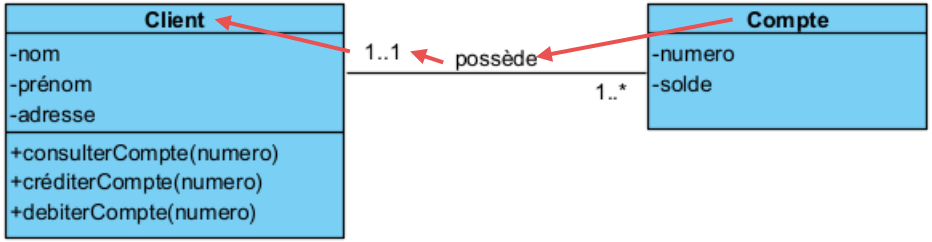


Cardinalités – sens de lecture

❑ Le **Client possède** de **1 à n Compte**



❑ Le **Compte possède un et un seul Client**



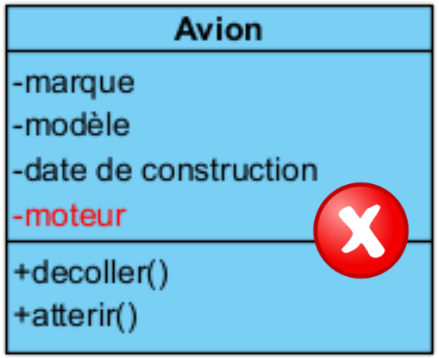
Concept de classe et d'association

❑ Diagrammes de classes

- Seules les propriétés correspondant à une information simple (date, chaine de caractère, nombre) apparaissent dans une classe, i.e. celles dont la structure n'a pas besoin d'être détaillée.
  - Nom
  - Prénom
  - Date de naissance, etc.

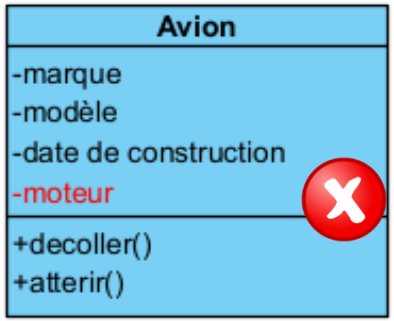
❑ Exemple de mauvaise modélisation

- Un moteur n'est pas un concept évident. Il possède de nombreuses propriétés (modèle, puissance, etc..).
- Un avion peut avoir plusieurs moteurs



Concept de classe et d'association

- Lorsqu'une propriété est complexe et doit être détaillée, on créé une classe.
- Exemple de bonne modélisation



- Pour les associations, ou **relations**, on utilise un segment de droite.

Il existe 3 types de relations

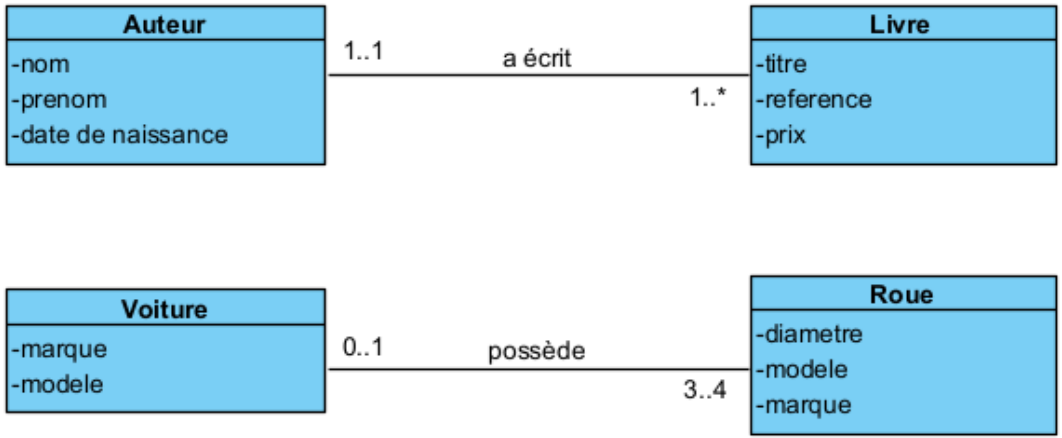
- Relation 1..n ou n..1
- Relation n..n
- Relation 1..1
- Ces types de relations décrivent la multiplicité bidirectionnelle.

La relation 1..n ou n..1

❑ Relation 1..n

- Cette relation formalise qu'une classe a une relation vers plusieurs occurrences d'une autre classe.

❑ Exemples



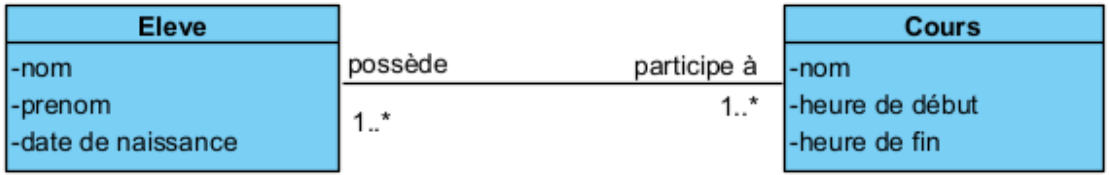
La relation n..n

❑ Relation n..n

- Elle formalise le fait que 2 classes ont des liens multiples et bidirectionnelles entre elles.

❑ Exemple

- Un élève participe a au moins un cours et en suivra sans doute beaucoup durant l'année
- Un cours va être suivi en général par au moins un élève mais on ne connait pas le nombre exact.



La relation 1..1

❑ Relation 1..1

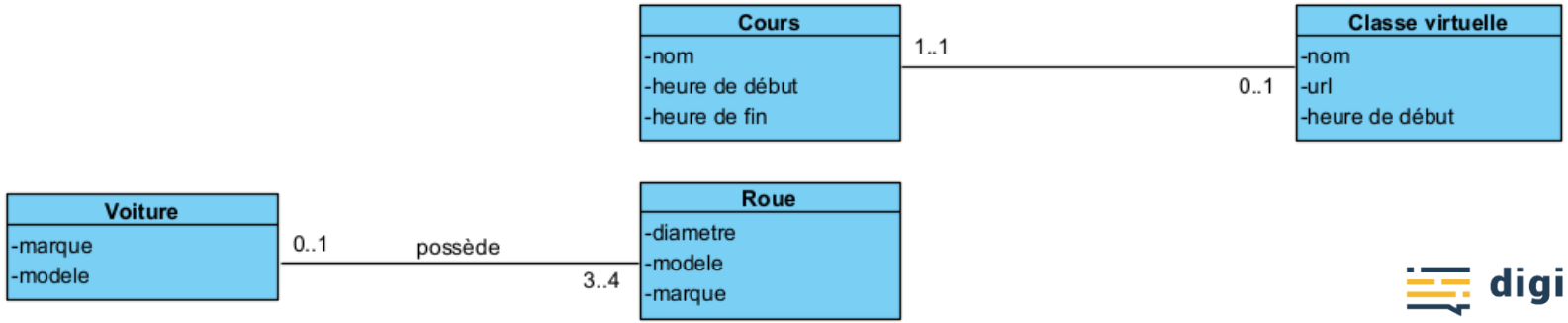
- Elle formalise une relation unique (non multiple) entre 2 classes.

❑ Exemple

- Un salarié possède **un unique** dossier RH (Ressources Humaines)
- Un dossier RH ne peut concerner qu'un **unique** salarié.

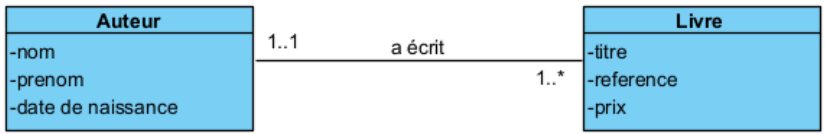
Peut-on avoir autre chose que des 1 et des n ?

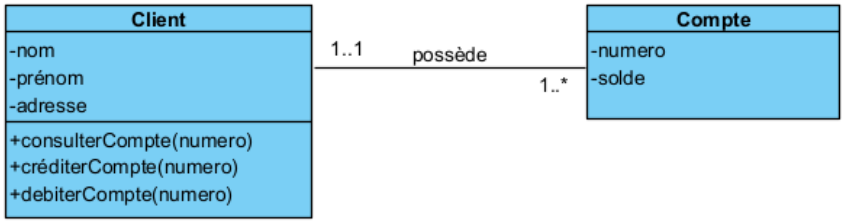
- **Oui**
  - Une relation a un **min** et un **max**
  - Ce min peut commencer à 0,1,2,3, etc...
  - Le max peut aller jusqu'à l'infini auquel cas on le note **n** ou *
- **Exemples**
  - Un salarié peut gérer de 0 à n salariés
  - Un cours peut se dérouler en classe virtuelle ou non



Un modèle exprime t'il la réalité ?

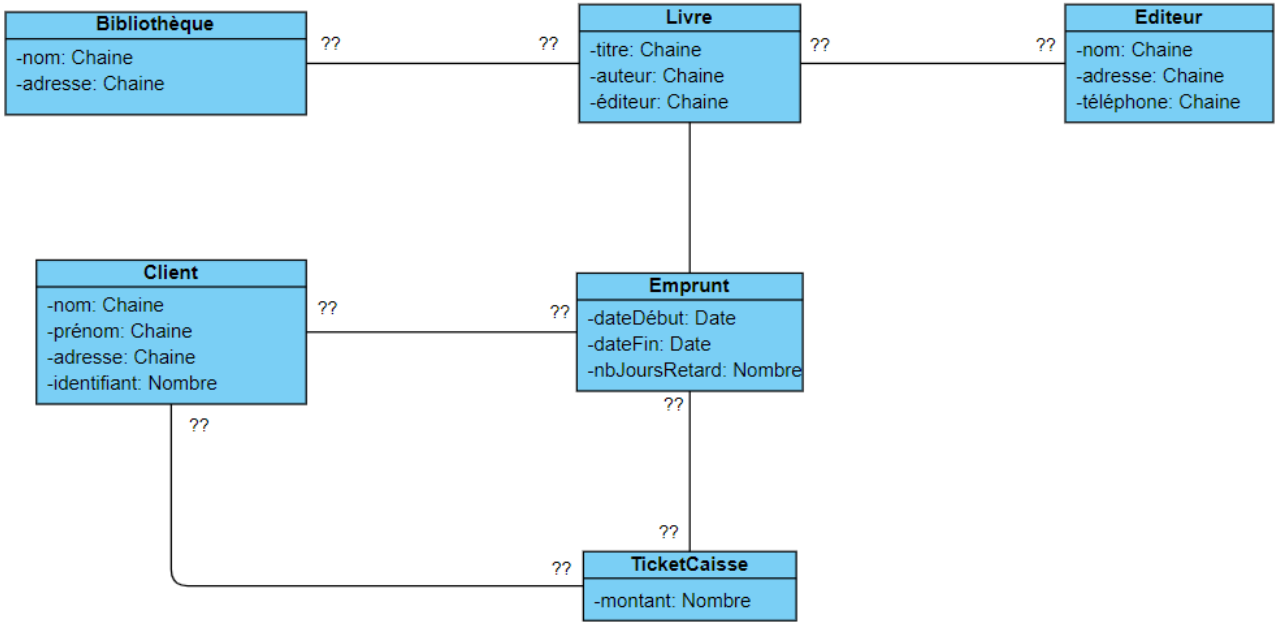
- Le modèle n'est pas la vérité. Il dépend du besoin exprimé par un client.
- Si on ne comprend pas bien le besoin du client, ou si le client n'exprime pas bien son besoin, cela peut avoir des conséquences graves.
- **Exemple :**
  - si on n'a pas prévu qu'un livre puisse avoir des co-auteurs.
  - Si on n'a pas prévu qu'un compte puisse appartenir à plusieurs personnes (compte joint)





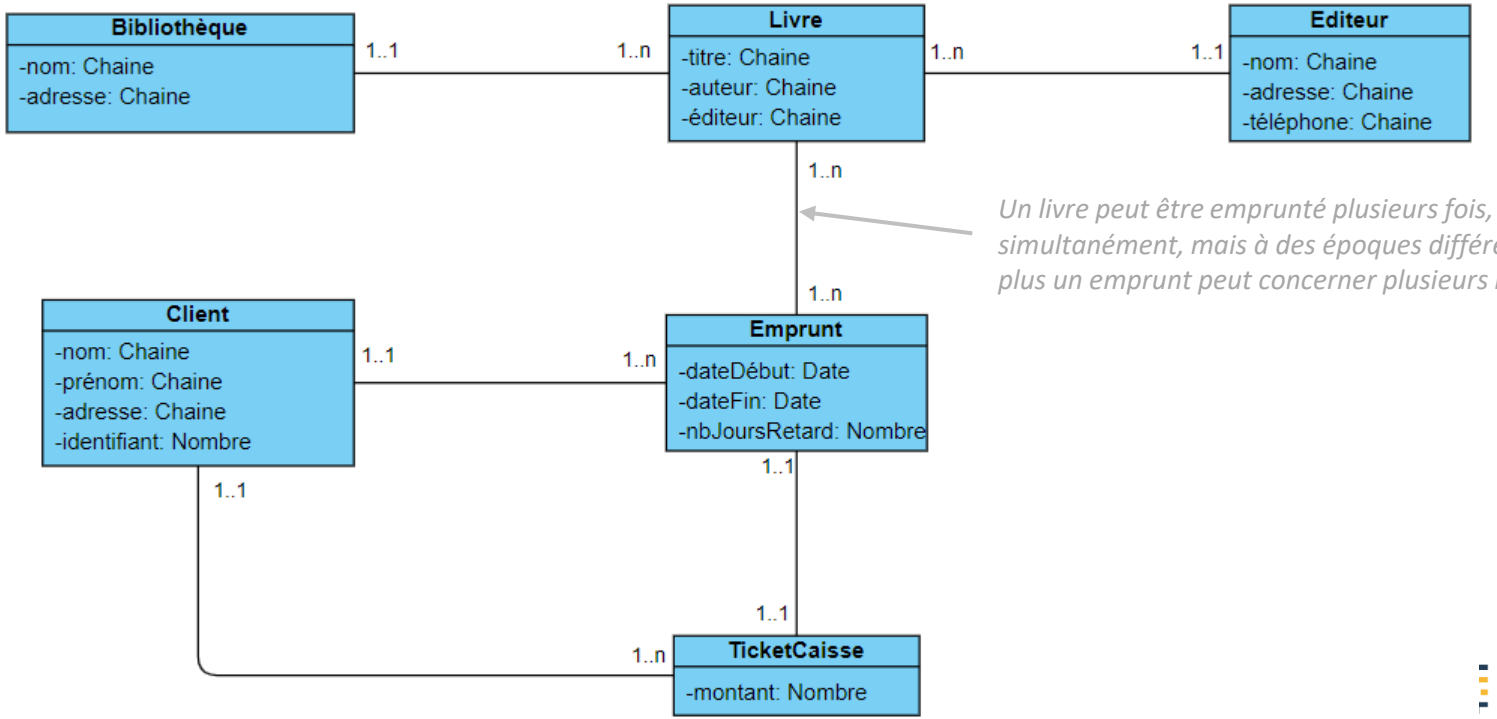
Exemple

- Voici le diagramme de classes qui décrit le fonctionnement d'une bibliothèque avec son mécanisme d'emprunts
- Quelles sont les cardinalités ?



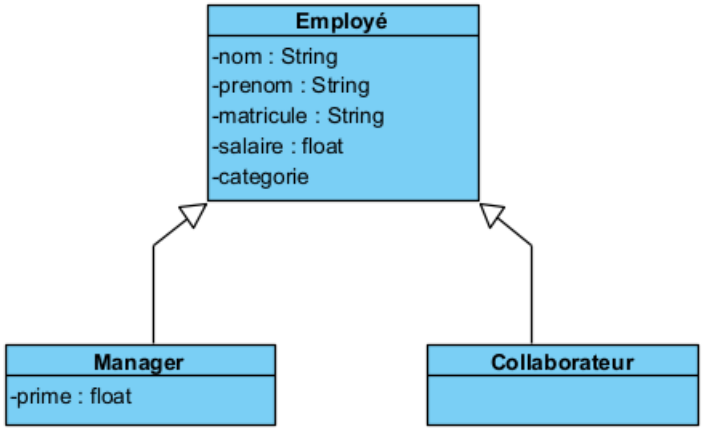
Corrigé

- Voici le diagramme de classes corrigé



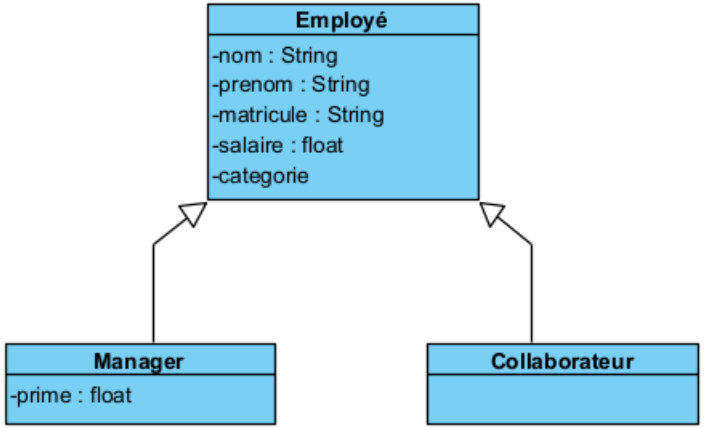
Généralisation dans un diagramme de classes

- La généralisation dans un diagramme de classes permet de factoriser des propriétés et des méthodes dans une classe parente.
- **Exemple :**
  - **Tous les employés** ont un matricule, un nom, un prénom, une adresse et un salaire.
  - Il existe cependant plusieurs types d'employés: les managers, les salariés ordinaires. Les managers ont par exemple en **propriété spécifique** une prime annuelle.



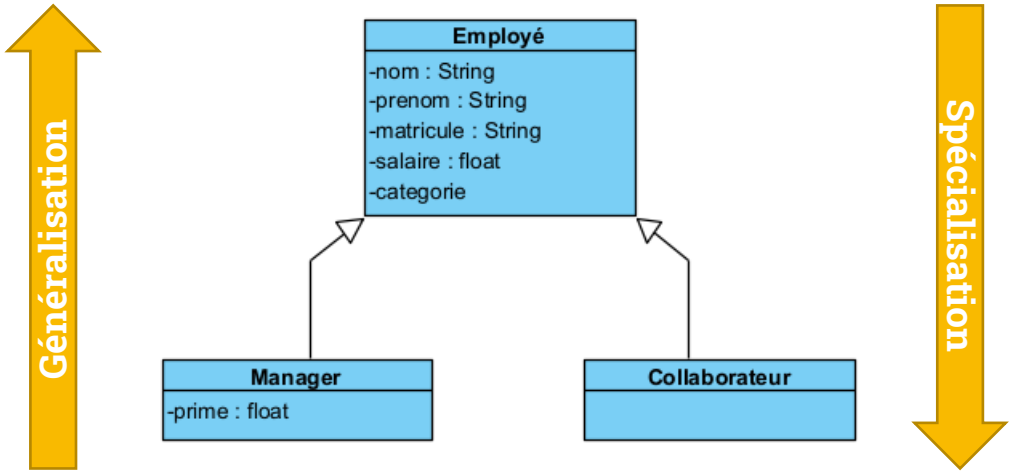
Généralisation dans un diagramme de classes

- La généralisation signifie que les classes filles (Manager et Salarié) possèdent toutes les propriétés de la classe mère (Employé).



Vocabulaire

- Le concept qui consiste à créer une classe mère pour centraliser les propriétés et méthodes communes s'appelle la **généralisation**
- Le concept qui consiste à créer des classes filles ne comportant que les propriétés et méthodes spécifiques s'appelle la **spécialisation**



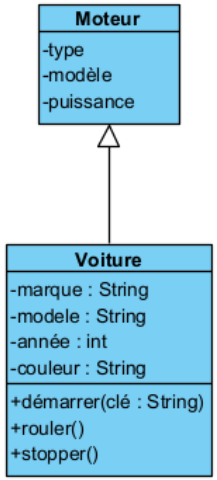
Association vs Généralisation (1/3)

- Pour savoir si une conception est correcte, utilisez le verbe
  - **Être** pour les **généralisations**
  - **Avoir** pour les **associations**

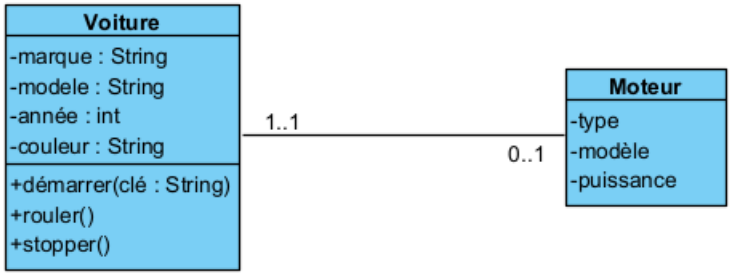
Association vs Généralisation (2/3)

- Est-ce que mon association ou ma généralisation est correcte ?
- Est-ce que je peux dire ?

Une voiture **est** un moteur ➔ généralisation



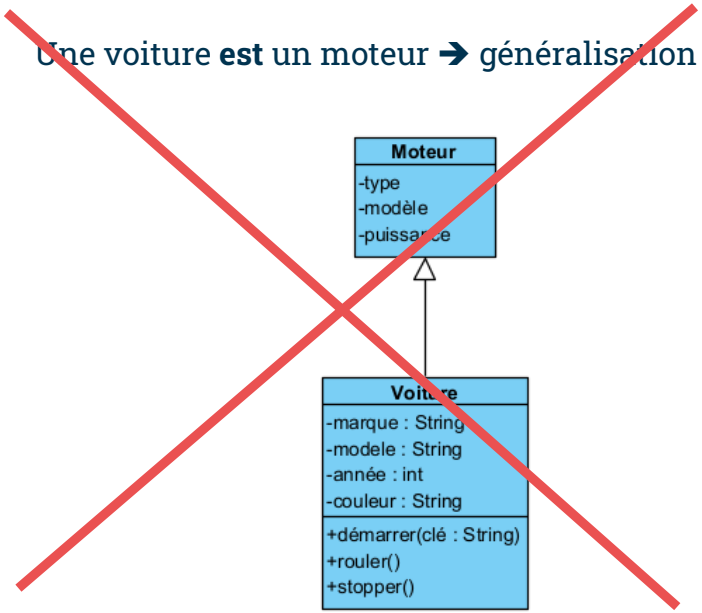
Une voiture **a** un moteur ➔ association



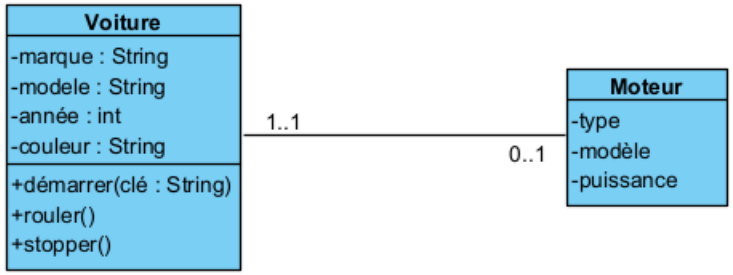
Association vs Généralisation (3/3)

- ❑ Est-ce que mon association est correcte ?
- ❑ Est-ce que je peux dire ?

Une voiture **est** un moteur ➔ généralisation



Une voiture **a** un moteur ➔ association



Réflexion sur la conception

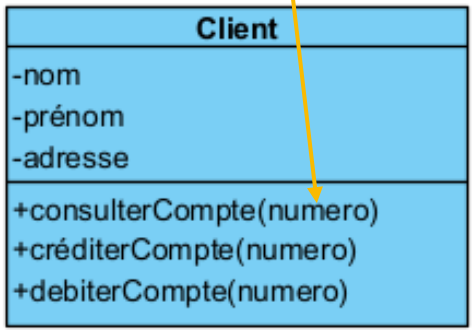
- Est-ce que je peux mettre en place une généralisation si ça m'arrange
- **Exemple** : pour hériter d'une ou plusieurs méthodes importantes situées dans une classe et dont j'ai besoin dans une classe fille ?

NON

- Je n'ai pas le droit de mettre en place une association ou un héritage si cela n'a pas de sens conceptuel. Seule exception: les classes techniques.

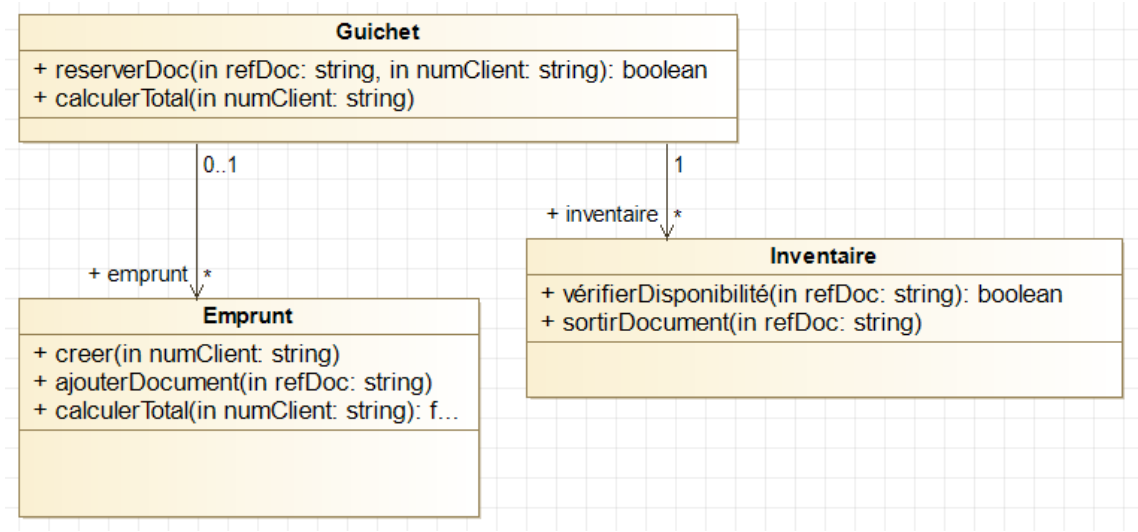
Comportement d'une classe

- En plus des propriétés, une classe peut posséder des **méthodes** qui décrivent ce que peut faire la classe.
- Les **méthodes** peuvent avoir des **paramètres**, c'est-à-dire des informations, qui leur sont fournies afin que la méthode fonctionne.
- Lorsqu'on ajoute une méthode à une classe, il faut imaginer de quoi elle va avoir besoin.
- Exemple:



Exemple de comportements

- Dans la classe **Guichet**, le comportement **reserverDoc** qui permet à un client de réserver un document exige en arguments:
  - le numéro du client
  - la référence de l'ouvrage.

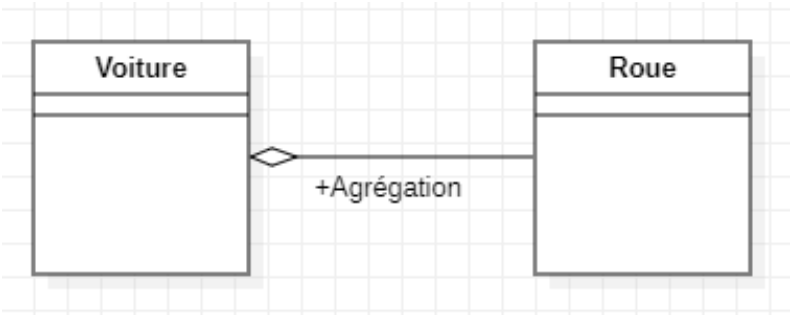


Différence entre association et dependency

- Une **association** implique qu'un objet "possède" une référence sur un autre objet (attribut), comme l'avion qui possède un ou plusieurs moteurs.
- Une **dependency** est une relation plus faible qu'une association. On dit qu'il y a une dépendance entre 2 objets si le premier prend le second en paramètre d'une méthode par exemple.

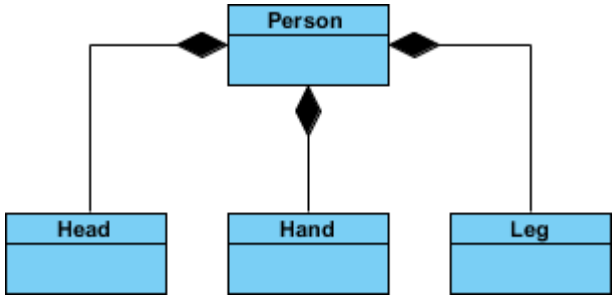
L'agrégation

- Une **agrégation** permet de préciser la nature de l'association entre 2 objets. Elle permet de réaliser un assemblage permettant de créer un objet plus complexe. L'agrégation définit que la voiture doit posséder des roues pour être fonctionnelle, intègre. La voiture, élément central, est appelé agrégat.
- **L'agrégé**, la roue, existe indépendamment de la Voiture. Si la voiture est détruite, la roue peut être réutilisée par exemple.



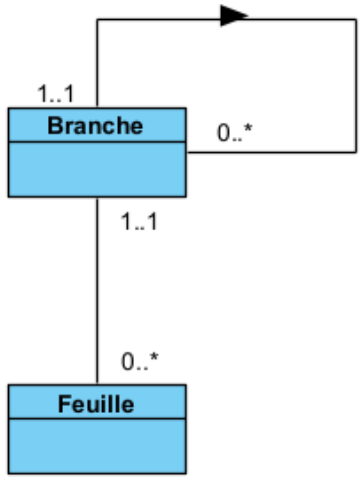
La composition

- La **composition** est plus forte que l'agrégation.
- Les vies du composant et des composés sont liées.



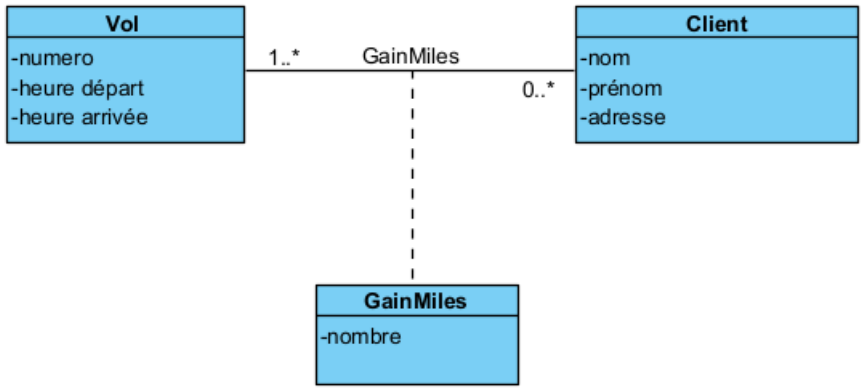
L'association réflexive

- L'association réflexive exprime qu'une classe peut "être associée" à plusieurs instances d'elle-même.



La classe d'association

- La classe d'association est une classe associée à une association de classes.
- Dans l'exemple ci-dessous le gain de points en miles est associé non pas au client, et non pas au vol mais au couple vol/client.



TP n°3

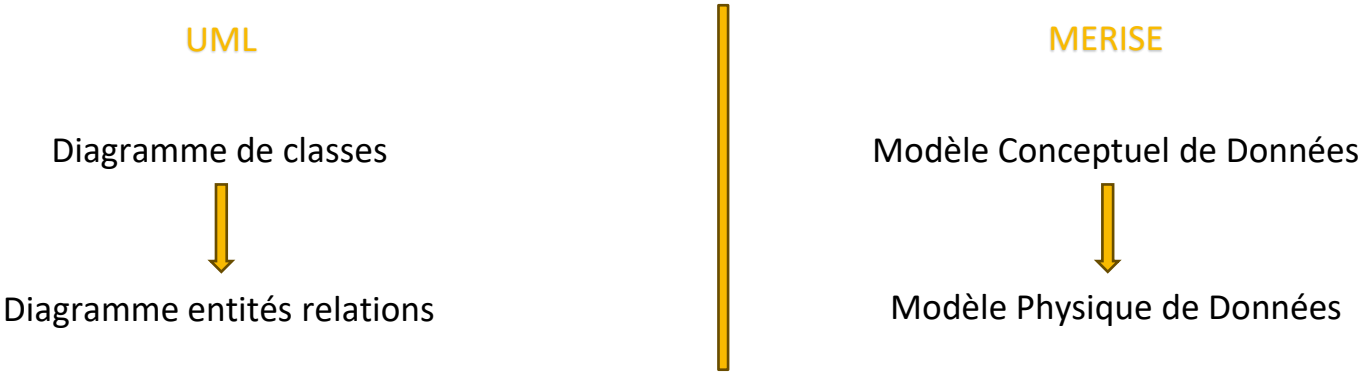
Réalisation d'un diagramme de classes

TP n°3: réalisez le diagramme de classes du collège

Diagramme entités-relations

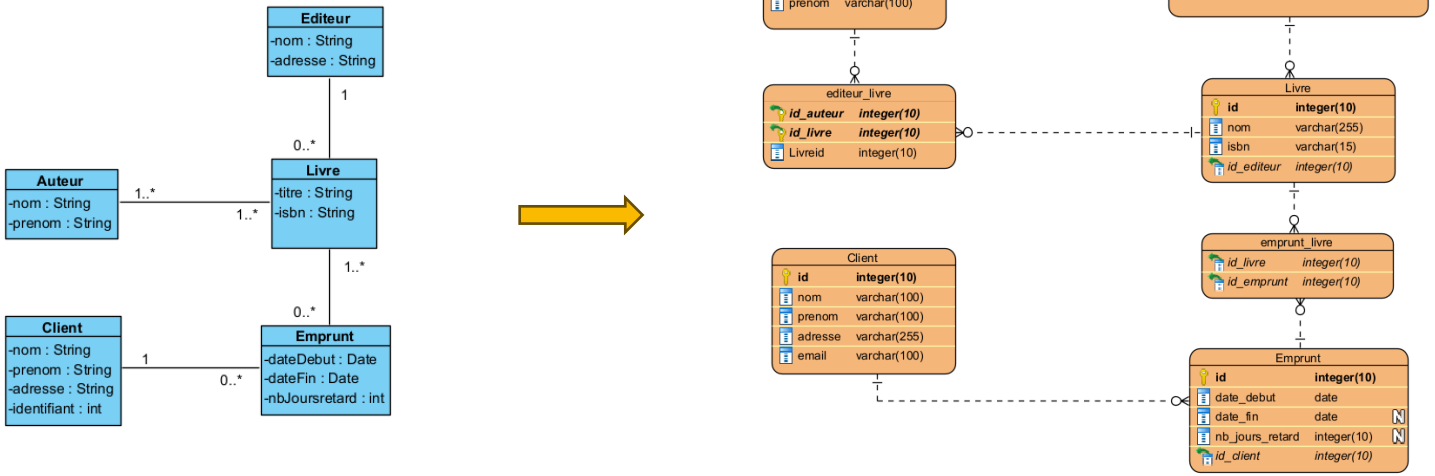
Le diagramme entités relations

- Le diagramme entités relations (ER) sert à modéliser une base de données.
- Il est réalisé après le diagramme de classes
- Dans la méthode Merise, on l'appelle Modèle Physique de données.
- En Merise, le MCD est l'équivalent du diagramme de classes



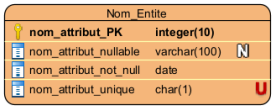
Le diagramme entités relations

- Le diagramme ER se déduit du diagramme de classes comme on va le voir dans la suite de ce cours.
- Il est donc indispensable d'avoir d'abord réalisé le diagramme de classes
- Exemple :



Les symboles

❑ Les symboles :



Entité avec ses attributs

Zéro ou Un

Un

Plusieurs

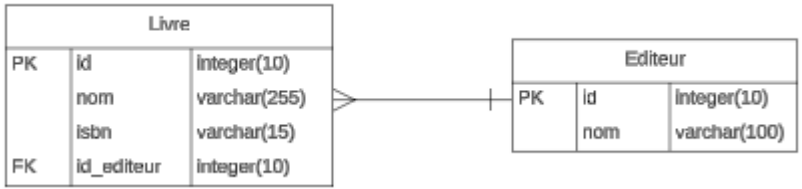
Zéro ou plusieurs

☞Les symboles utilisés dans l'entité pour représenter : les clés primaires, les colonnes associées à une clé étrangère, ou encore les contraintes, sont libres.

Les symboles - exemples

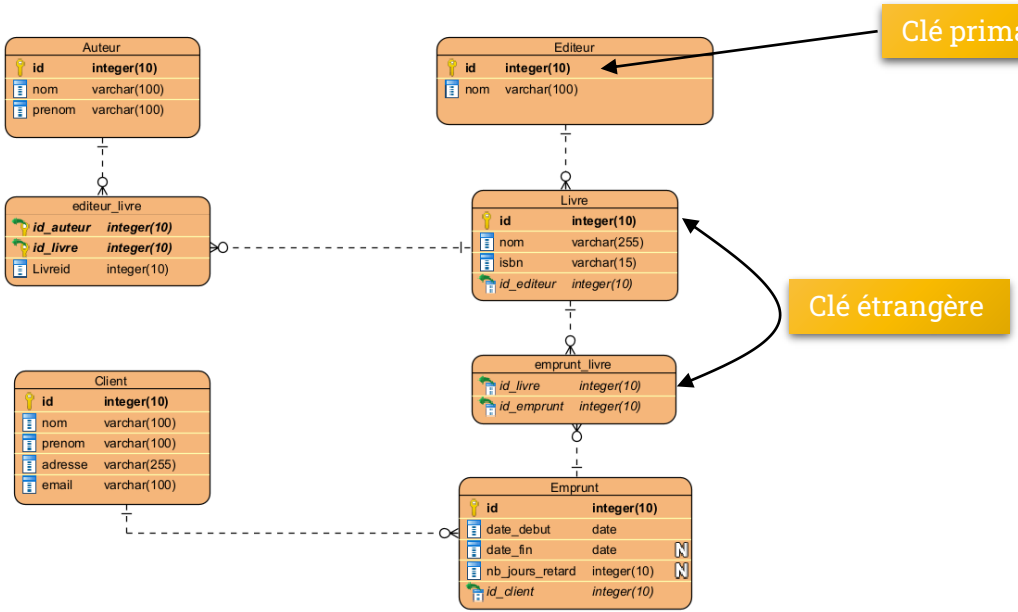
- VisualParadigm :

- LucidChart :



Le diagramme entités relations

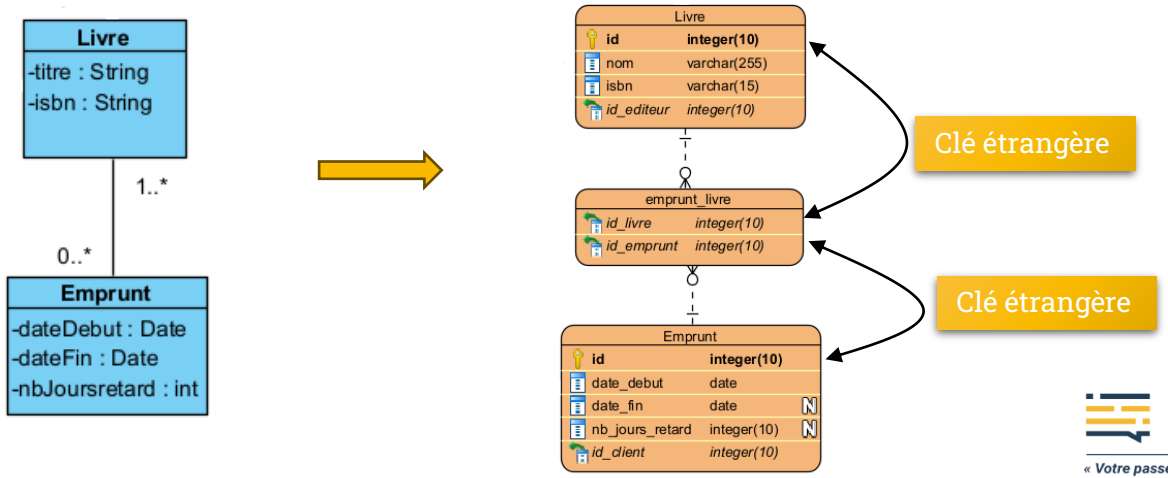
- Dans le diagramme entités relations, les entités représentent les futures tables (au sens base de données relationnelles) et les relations les futures clés étrangères.



Les règles de passage: relation "Many to Many"

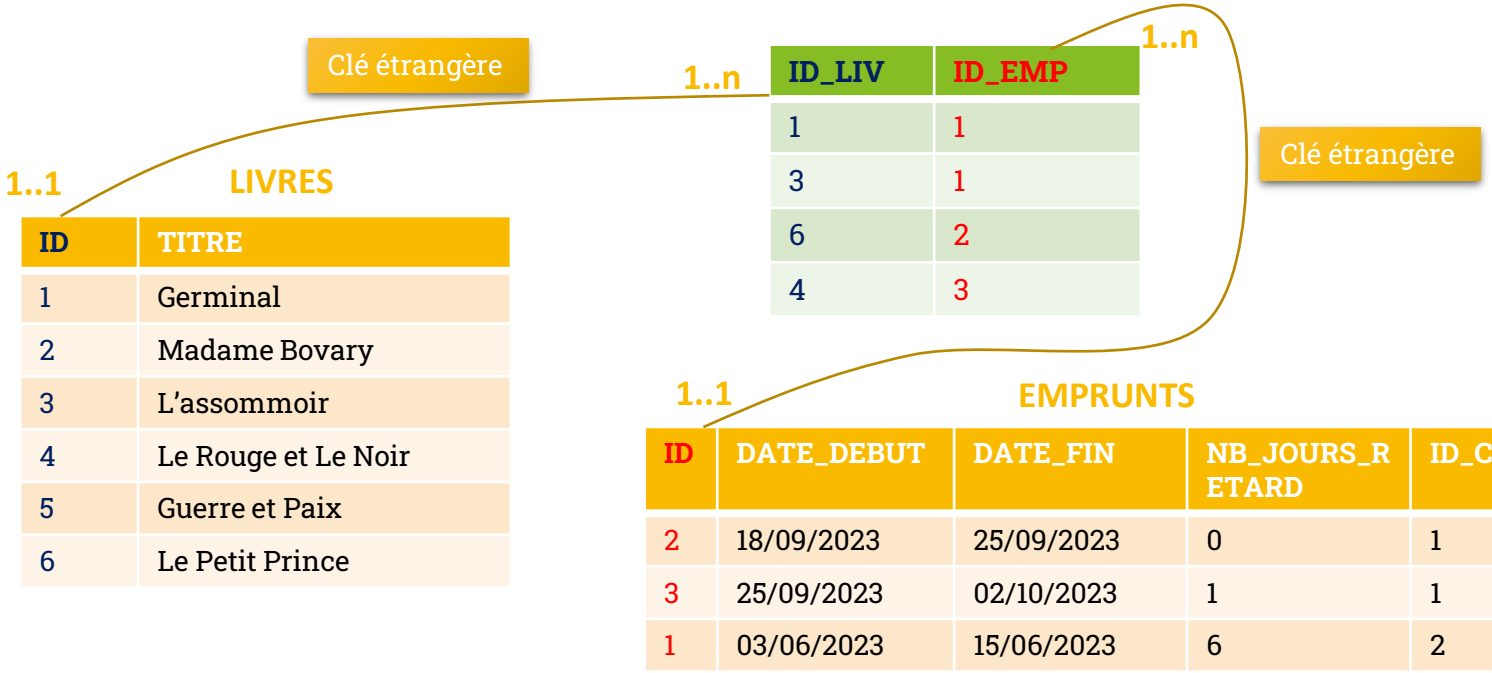
- Une relation **Many to Many** dans le diagramme de classes devient une **table de jointure** avec **2 clés étrangères.**

- Exemple entre Emprunt et Livre :
  - Une table de jointure **emprunt_livre**
  - Une clé étrangère entre Livre (id) et emprunt_livre (id_livre)
  - Une clé étrangère entre Emprunt (id) et emprunt_livre (id_emprunt)



Exemple de relation Many to Many implémentée dans une base de données

- Exemple avec des tables



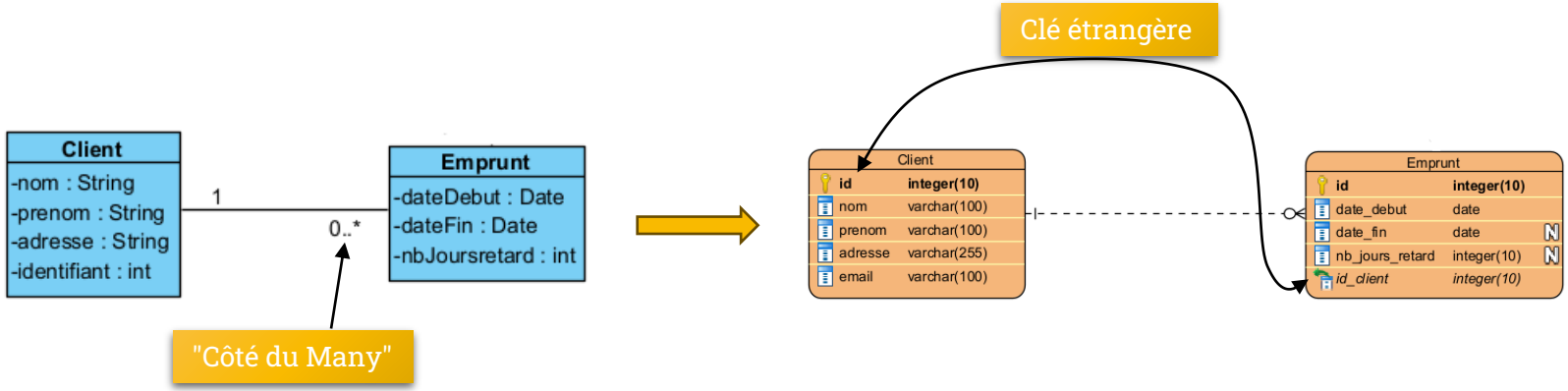
ID_LIV	ID_EMP
1	1
3	1
6	2
4	3

ID	TITRE
1	Germinal
2	Madame Bovary
3	L'assommoir
4	Le Rouge et Le Noir
5	Guerre et Paix
6	Le Petit Prince

ID	DATE_DEBUT	DATE_FIN	NB_JOURS_RETARD	ID_CLIENT
2	18/09/2023	25/09/2023	0	1
3	25/09/2023	02/10/2023	1	1
1	03/06/2023	15/06/2023	6	2

Les règles de passage: relation One to Many

- Une relation **One to Many** devient une **clé étrangère** du "côté du Many"
- Exemple entre Client et Emprunt :
  - Est-ce qu'on met un attribut id_client dans Emprunt ou un attribut id_emprunt dans Client ?
  - Un client peut avoir plusieurs emprunts (de 0 à n), le côté du Many est donc côté Emprunt.
  - C'est donc dans **Emprunt** qu'on ajoute un attribut **id_client**



Exemple de relation One to Many implémentée dans une base de données

- Exemple avec des tables

EMPRUNTS

ID	DATE_DEBUT	DATE_FIN	NB_JOURS_RETARD	ID_CLIENT
2	18/09/2023	25/09/2023	0	1
3	25/09/2023	02/10/2023	1	1
1	03/06/2023	15/06/2023	6	2

1..n

1..1

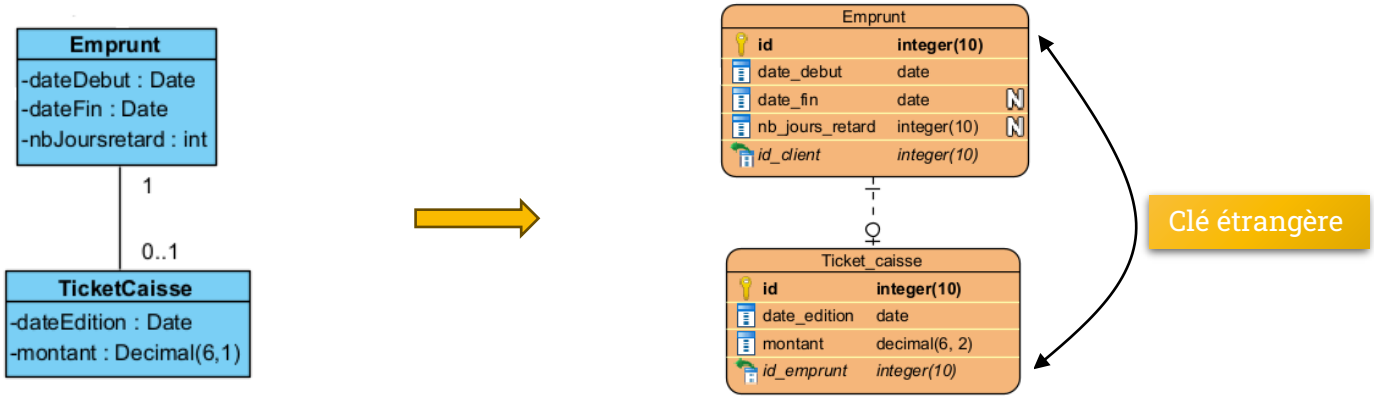
Clé étrangère

CLIENTS

ID	NOM
1	DOTENET
2	PITON
3	GEAVAS

Les règles de passage: relation One to One

- Une relation **One to One** devient une clé étrangère du côté le plus pertinent
- Exemple entre Emprunt et TicketCaisse :
  - On peut par exemple mettre un attribut **id_emprunt** dans l'entité TicketCaisse



Relation 1..1

- Exemple avec des tables :

Emprunt

ID.	Date de début	Date de fin	Nb jours de retard	Id. client
37	12/06/2023	26/06/2023	2	11
48	15/06/2023	29/06/2023	0	2
59	17/06/2023	23/06/2023	0	7

1..1

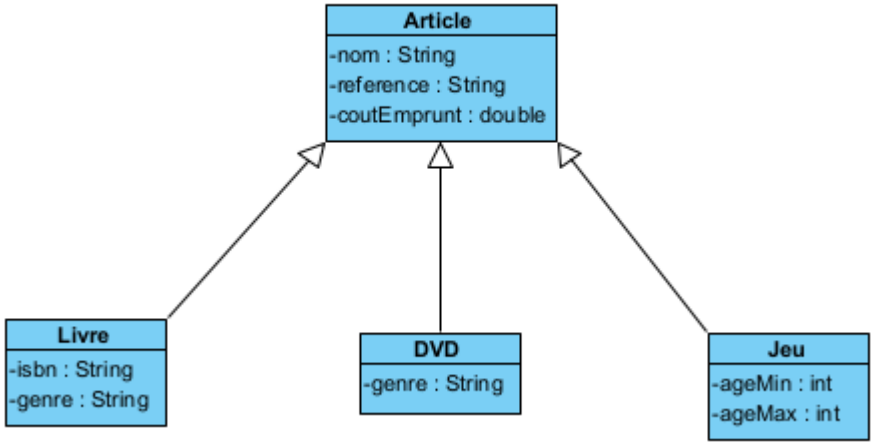
1..1

Ticket Caisse

Id	Date édition	Montant	Id. emprunt
112	26/06/2023	5,5	**37**
129	29/06/2023	5	**48**
139	23/06/2023	2,5	**59**

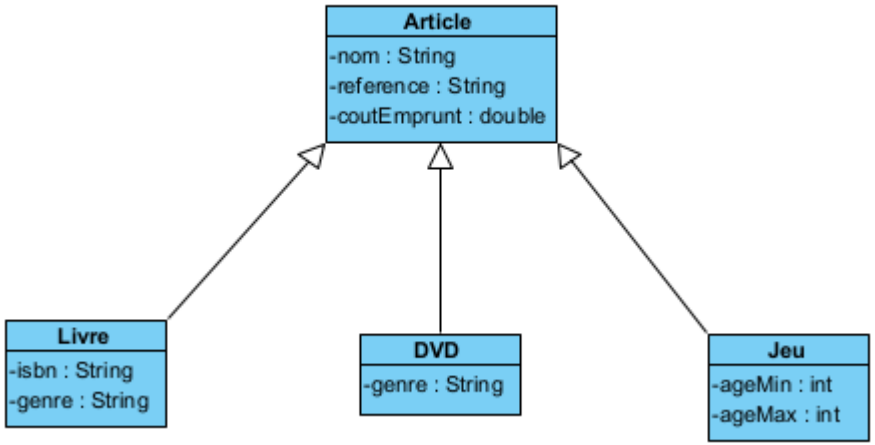
Les règles de passage: généralisation

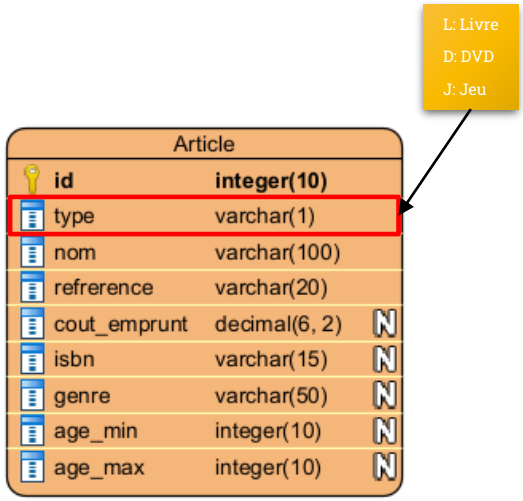
- La **généralisation** et son contraire, la **spécialisation** (ou héritage), sont des éléments importants des diagrammes de classes.
- Dans un **diagramme ER** il existe plusieurs manières d'implémenter un héritage.
- Exemple de généralisation dans un diagramme de classes :
  - Article est une généralisation de Livre, DVD et Jeu
  - Livre, DVD et Jeu sont des spécialisations d'Article



Implémentation entité unique

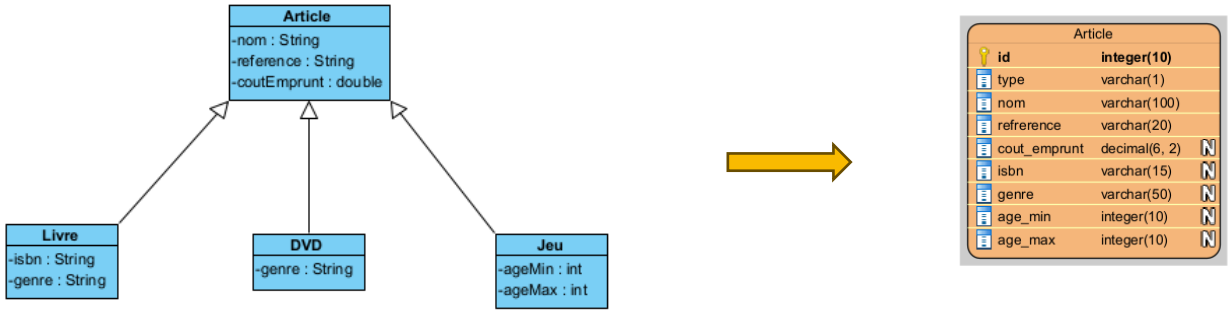
- Une première solution est l'entité unique
- Dans le cas de l'entité unique (table unique) il va falloir différencier les données avec un attribut **technique** type (non issu de considérations métier).
- Exemple:





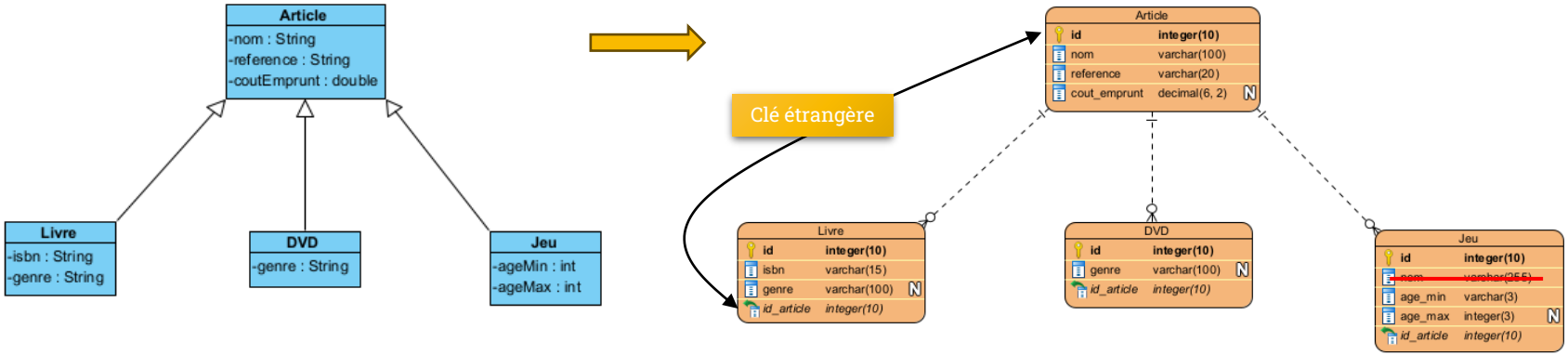
Implémentation entité unique – avantages / inconvénients

- Avantages:
  - Simplicité
  - Performance
- Inconvénients:
  - La plupart des colonnes sont **nullables** car en fonction du type de l'article (Livre, DVD ou Jeu), certaines données sont renseignées ou non.
  - Impossible de mettre des contraintes d'intégrité sur la plupart des colonnes.



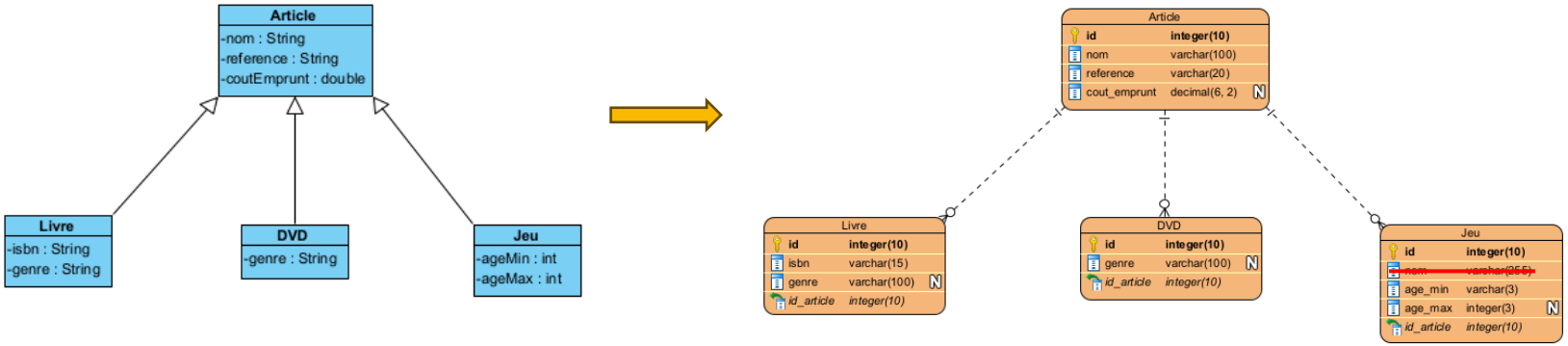
Implémentation une entité par classe

- Une seconde solution est d'avoir **une entité par classe**.
- Dans ce cas, les données sont systématiquement réparties dans 2 tables distinctes reliées par une clé étrangère.
- Exemple :



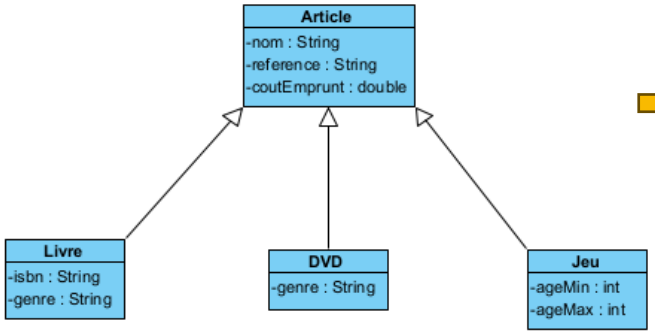
Implémentation une entité par classe – avantages / inconvénients

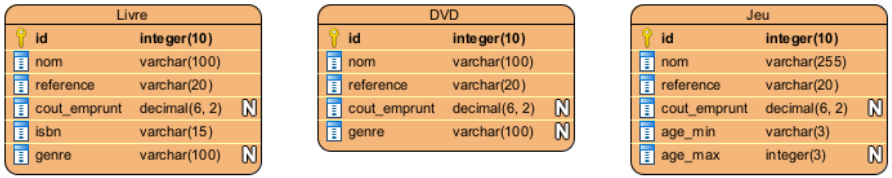
- Avantages:
  - Possibilité de mise en place de contraintes d'intégrité sur les données
  - Distinction claire des différents types de données
- Inconvénients:
  - Moins performant : jointure systématique



Implémentation une entité par classe spécialisée

- La dernière solution est **une entité par classe spécialisée**
- Dans ce cas, la classe Article n'est pas représentée par une entité.
- Seules les classes filles sont représentées par une entité.
- Exemple :





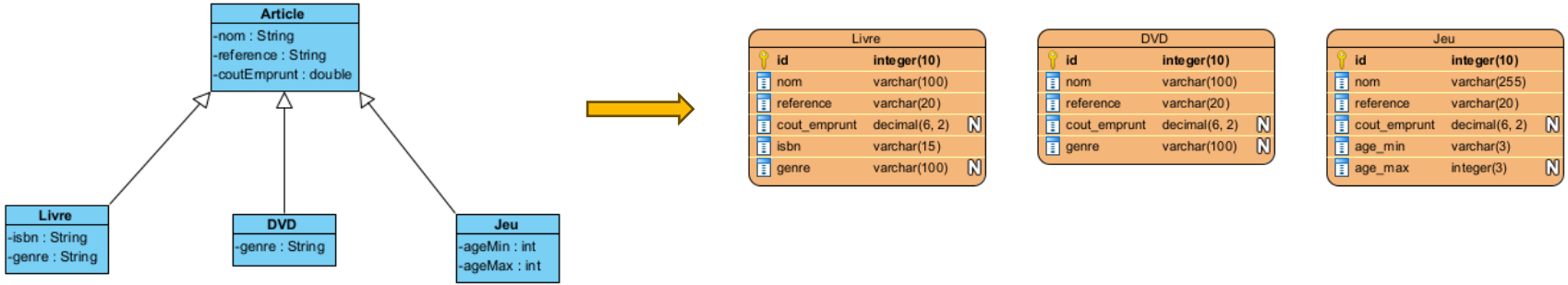
Implémentation une entité par classe spécialisée – avantages / inconvénients

❑ Avantages:

- Possibilité de mise en place de contraintes d'intégrité sur les données
- Distinction claire des différents types

❑ Inconvénients:

- Redondance d'attributs: si on ajoute un attribut dans la classe Article (diagramme de classes), on doit créer un attribut par entité au niveau du diagramme ER => risques d'oubli.
- On perd de l'information au niveau conceptuel. On perd l'information de spécialisation.



TP n°4

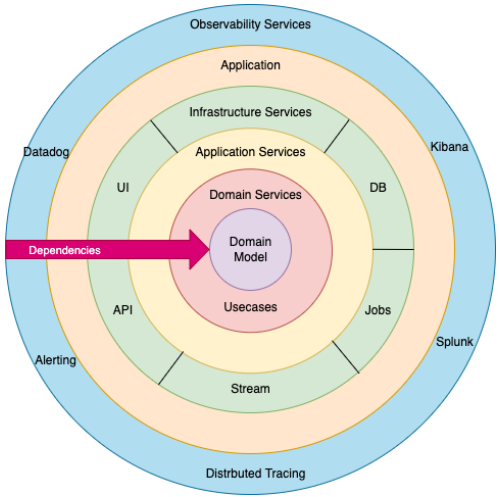
Réalisation d'un diagramme entités relations

TP n°4: réalisez le diagramme entités relations du collège

Découpage en couches et bonnes pratiques de conception

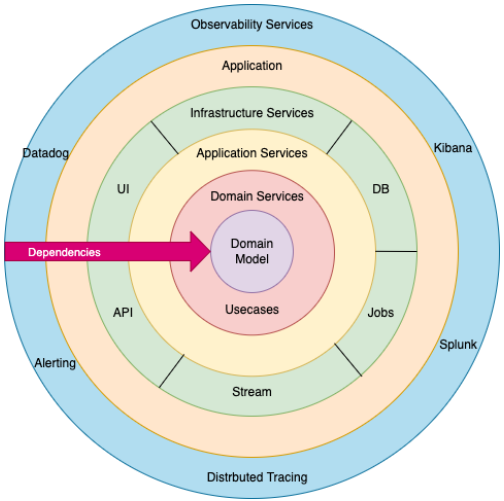
Introduction

- Le découpage en couches permet de bien structurer son application
- Les avantages sont multiples : évolutivité, maintenabilité, robustesse, testabilité
- L'architecture en couches modèle, ou en oignions :



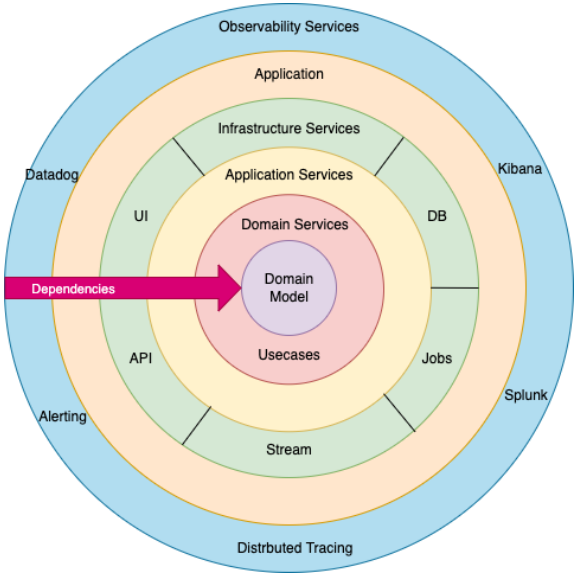
Observability services

- C'est la couche responsable du **monitoring** de l'application.
- C'est une couche extérieure à l'application et constituée d'outils pour surveiller et alerter si besoin.
- Elle peut également contenir des outils pour analyser et comprendre des problèmes de vie courante (bugs, piratage, etc..)



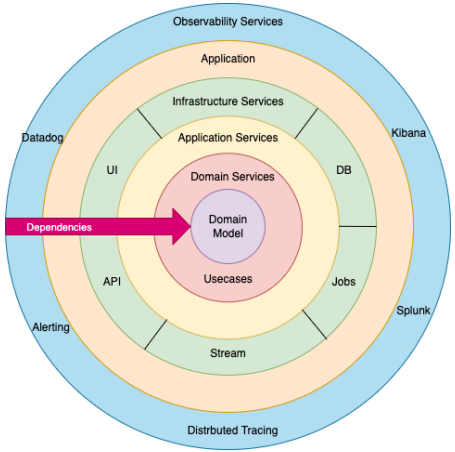
Infrastructure services

- C'est la couche à laquelle appartient les contrôleurs mais également les classes qui permettent l'accès à des ressources comme la base de données, des jobs, etc.
- C'est une couche technique, i.e. non-métier mais indispensable.



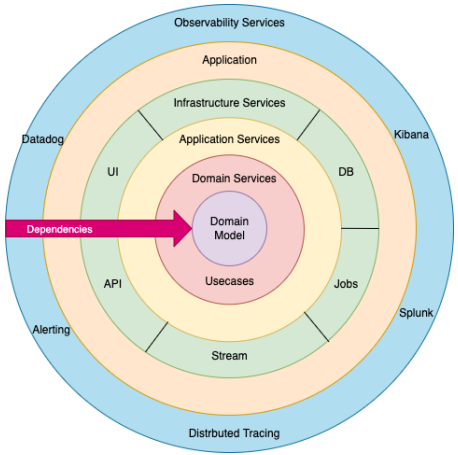
Application services

- C'est la couche responsable de la réalisation d'un cas d'utilisation, par exemple "créer un nouvel élève".
- Il y a une classe de type "Application services" par cas d'utilisation
- C'est ce type de classe qui orchestre les différents services afin de réaliser le cas d'utilisation.



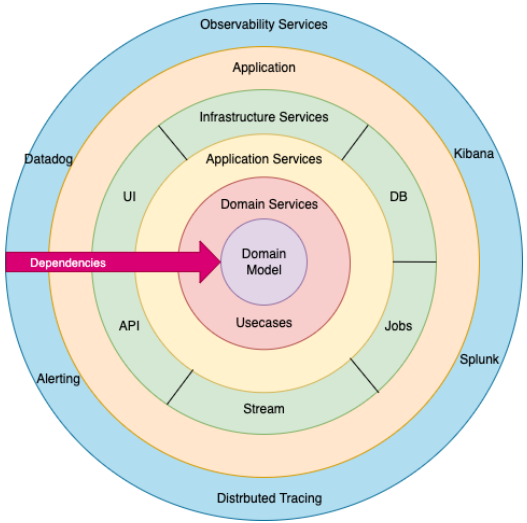
Domain services

- C'est la couche responsable de la vérification des règles métier et qui lève une erreur si une règle métier n'est pas respectée. Exemple: le nom de l'élève n'est pas renseigné.
- En général il y a un service par entité métier : EleveService, ClasseService, BulletinService, etc.
- La couche **Application Services** utilise une ou plusieurs classes de type **Domain services** pour les contrôles métier.



Domain model

- Ce sont les entités métier : Eleve, Classe, Bulletin, etc.
- Les entités métier n'ont aucune dépendance technique à l'exception des annotations qui sont tolérées. Les annotations permettent notamment de configurer la couche ORM.

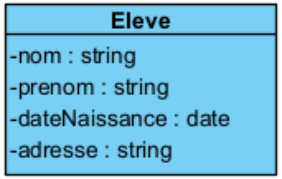


Les échanges de données avec l'extérieur

- Aujourd'hui les échanges avec le front, ou des applications externes, utilisent principalement JSON ou XML.
- Une classe de type contrôleur (infrastructure services) échange des données avec l'extérieur au format JSON.
- En général un contrôleur ne reçoit pas directement un message JSON à traiter.
- Des librairies, comme Jackson (filière Java), permettent de transformer le message JSON en instance de classe, si bien qu'un contrôleur va recevoir en paramètre non pas une chaine de caractère mais une instance d'objet.
- Une mauvaise pratique est d'utiliser une entité métier, par exemple Eleve, pour mapper le message JSON.
- Une bonne pratique est d'utiliser une classe EleveModel ou EleveDto pour mapper le message JSON. Dans la couche **application services**, l'instance d'EleveModel est transformée en instance d'Eleve.

Exemple Eleve vs EleveDto

- Supposons qu'on ait l'entité métier Eleve ci-dessous :



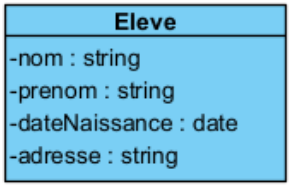
- Et supposons également qu'on ait besoin d'afficher une liste d'élèves côté front avec l'âge et un format d'affichage de la date de naissance particulier :

Nom	Prénom	Date de naissance	Age	Adresse
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...

- Problèmes :
  - Qui calcule l'âge ? Ça ne va pas se faire tout seul !
  - Qui doit formater la date au format JJ/MM/AAAA ? Ça ne va pas se faire tout seul !

Exemple Eleve vs EleveDto

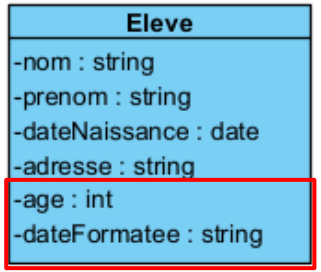
Nom	Prénom	Date de naissance	Age	Adresse
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...



- Solution 1:
  - on envoie au front une liste d'élèves et le front se débrouille
- Problèmes:
  - Le front devient de facto un expert en calcul des âges, ce qui ne doit pas être une responsabilité du front
  - Le front devient également de facto un expert en formatage des dates. Pourquoi pas mais il n'a pas la connaissance qu'a le back sur les préférences de l'utilisateur par exemple.

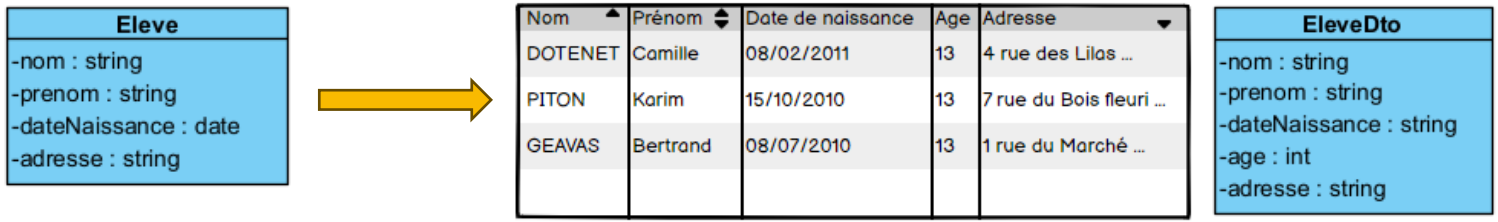
Exemple Eleve vs EleveModel

Nom	Prénom	Date de naissance	Age	Adresse
DOTENET	Camille	08/02/2011	13	4 rue des Lilas ...
PITON	Karim	15/10/2010	13	7 rue du Bois fleuri ...
GEAVAS	Bertrand	08/07/2010	13	1 rue du Marché ...



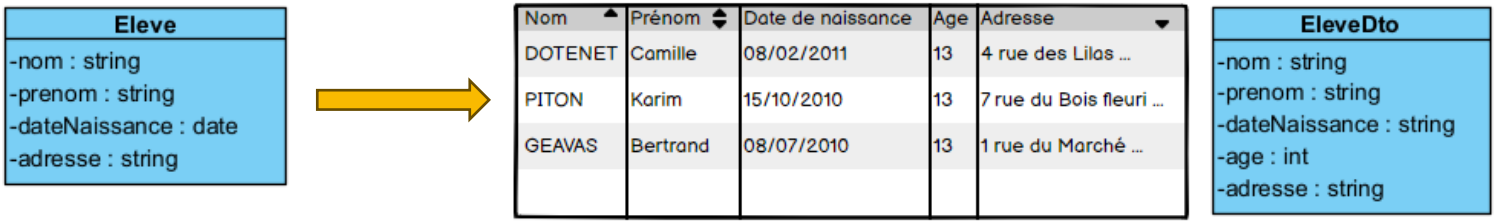
- Solution 2:
  - on ajoute à l'entité métier **Eleve** 2 attributs: un attribut age et un attribut "date formatée"
  - On réalise côté back le calcul de l'âge et le formatage de la date en tenant compte des préférences de l'utilisateur.
- Problèmes:
  - On commet un "crime contre le métier"
  - Le métier doit être indépendant des contraintes techniques et de présentation.

Exemple Eleve vs EleveDto



- Solution 3:
  - On créé une classe EleveModel, ou EleveDto qui va avoir exactement les données attendues par la vue.
- Problèmes:
  - Ça donne un peu plus de travail au départ mais c'est très facile à faire évoluer en fonction des besoins des vues.
  - Il faut maintenir toute une zoologie de Dtos.

Conclusion



- De toutes les solutions c'est la solution 3 qui est **actuellement** considérée comme la meilleure pratique même si elle n'est pas exempte de défauts.

Dependency Inversion Principle

Introduction

- Le principe **Dependency Inversion Principle** est un principe très important.
- C'est celui qui explique l'organisation des couches dans Spring Boot par exemple
- Ce principe dit qu'il faut éviter au maximum les dépendances entre classes concrètes.
- Exemple qui viole le DIP:

```
class Ampoule {

   public void allumer() {
   }

   public void eteindre() {
   }
}
```

```
class Interrupteur {

   private Ampoule ampoule;

   public Interrupteur(Ampoule ampoule) {
      this.ampoule = ampoule;
   }

   public void appuyer() {
      if (ampoule.isOn()) {
         ampoule.eteindre();
      } else {
         ampoule.allumer();
      }
   }
}
```

Exemple

- **Exemple qui respecte le DIP** : pour respecter le DIP il est nécessaire d'ajouter une abstraction entre Interrupteur et Ampoule sous la forme d'une interface

```
interface Allumable {

   void allumer();

   void eteindre();
}
```

```
class Ampoule implements Allumable {

   @Override
   public void allumer() {
   }


   @Override
   public void eteindre() {
   }
}
```

```
class Interrupteur {
   private Allumable allumable;


   public Interrupteur(Allumable allumable) {
      this.allumable = allumable;
   }


   public void appuyer() {
      if (allumable.isOn()) {
         allumable.eteindre();
      } else {
         allumable.allumer();
      }
   }
}
```

Intérêt du DIP pour les tests

- L'intérêt du DIP ne se situe pas réellement au niveau des entités métier mais plutôt au niveau des classes de service.
- C'est la raison pour laquelle ce principe est à la base de Spring.
- L'intérêt est de faciliter le remplacement d'une implémentation par une autre et de grandement faciliter la testabilité via l'injection de **mocks**.
- Imaginons qu'une **classe PersonneService** ait besoin d'une **classe PersonneJpa** pour accéder en base de données.
  - Je ne peux pas créer de tests unitaires de PersonneService sans accéder à la base de données puisqu'il y a une dépendance forte entre les 2 classes.
- Imaginons maintenant que la classe PersonneService ait une référence sur une **interface** PersonneRepository pour accéder en base de données.
  - Je peux facilement injecter une implémentation PersonneRepositoryTest qui n'accède pas à la base de données dans cette classe.

Exemple pour le test (1/2)

```java
@Service
public class PersonneServiceImpl implements PersonneService {

   private PersonneRepository personneRepository;

   public PersonneServiceImpl(PersonneRepository personneRepository) {
      this.personneRepository = personneRepository;
   }

   @Override
   public PersonneAvecRoleDto findbyId(Long id) {
      Optional<Personne> opt = personneRepository.findById(id);
      if (opt.isPresent()) {
         return new PersonneAvecRoleDto(opt.get());
      }
      return null;
   }
}
```

Spring injecte
L'implémentation par défaut qui est
PersonneRepositoryImpl

```java
public interface PersonneRepository {

   Optional<Personne> findById(String idI);
}
```

```java
@Service
public class PersonneRepositoryImpl implements
PersonneRepository {

   public Optional<Personne> findById(String idI){
   }
}
```

Exemple pour le test (2/2)

Pour les tests unitaires, injection d'un mock qui est une classe qui implémente l'interface mais n'accède pas à la bdd.

```
public class PersonneServiceImplTest {

   @Test
   public void testFindById() {
      PersonneServiceImpl service = new PersonneServiceImpl(new PersonneRepositoryTest());

   }
}
```

```
public class PersonneRepositoryTest implements PersonneRepository {

   @Override
   public Optional<Personne> findById(Long id) {
      return Optional.empty();
   }
}
```

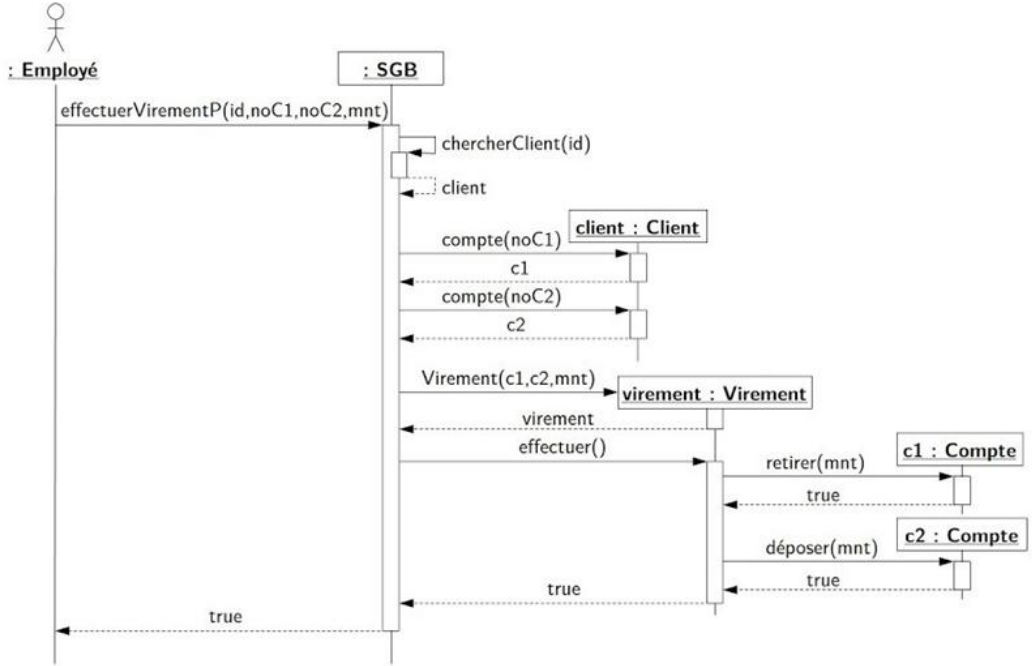
Diagramme de séquence

Diagramme de séquences

- Il fait partie des diagrammes de comportement (Behavioral diagrams)
- Dans un diagramme de séquences, des messages sont transmis entre objets.
- Avec le diagramme de classes, on cherche à représenter un algorithme :
  - Permet de représenter un algorithme complexe pour l'équipe de développement
  - On est un peu plus proche de la programmation
- Contraintes :
  - Avoir des diagrammes de classes très complet avec les méthodes

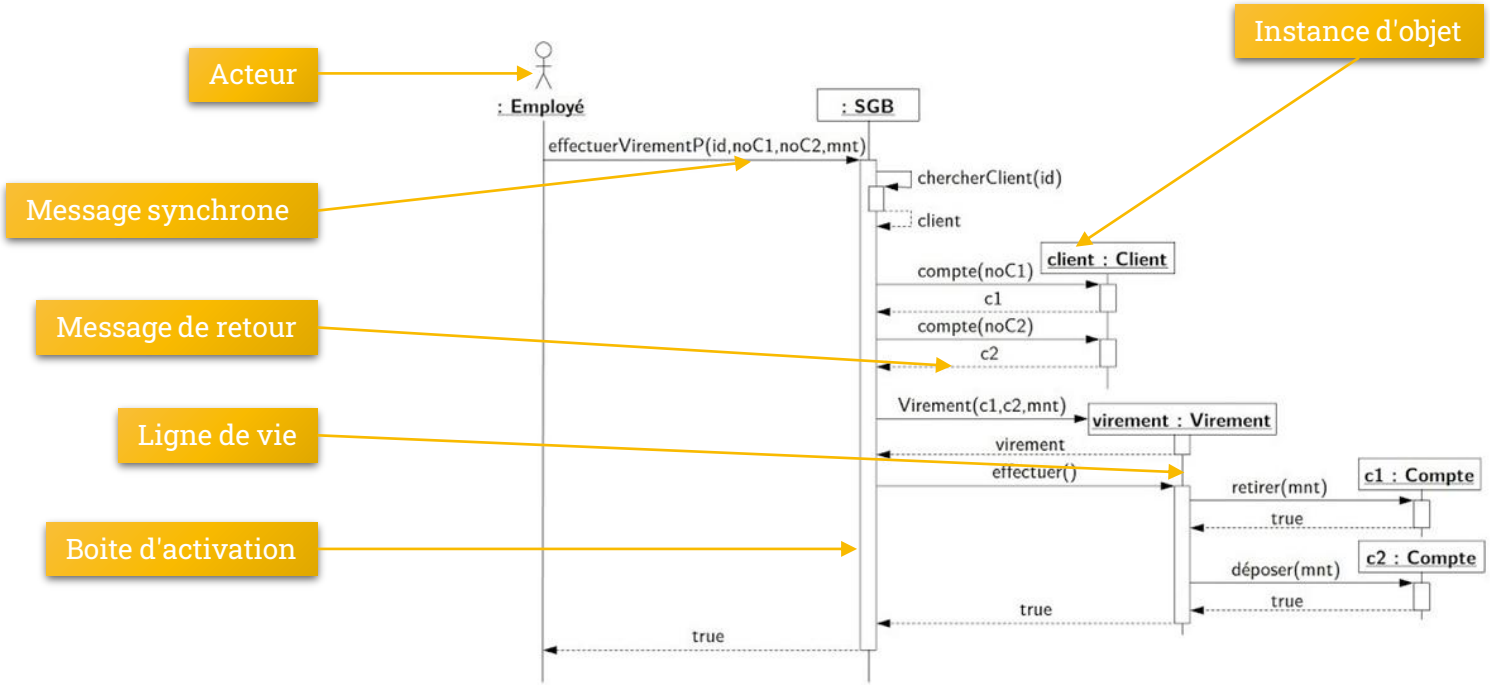
Diagramme de séquences

- Exemple



Les symboles

- Les symboles



Explications

- Explications concernant les symboles :
  - Un acteur initie la séquence
  - Un message synchrone est un terme générique qui peut désigner un appel de méthode ou de fonction avec des paramètres.
  - Un message de retour correspond à un retour de méthode ou de fonction par exemple
  - Une ligne de vie correspond à la vie de l'instance d'objet
  - Une boite d'activation correspond au temps que consacre une instance d'objet à effectuer une tâche.

Les symboles complémentaires

- Les autres symboles :

Message asynchrone

Boucle avec condition tant que

Condition avec alternative

TP n°4

Réalisation d'un diagramme de séquences

TP n°4: réalisez le diagramme de séquences de connexion à l'application

Introduction au Model Driven Development

Introduction

- Le Model Driven Development est complémentaire à UML, elle ne le remplace pas.
- Le MDD ajoute des étapes à la conception traditionnelle afin de générer du code à partir d'autres modèles.
- L'objectif ultime est de produire du code automatiquement à partir des modèles. On ne touche plus au code directement.
- L'abstraction et l'automatisation sont au cœur de l'approche.
- Des outils de transformation sont nécessaires pour générer le code.
- Le MDD est une démarche qui **ne fait pas l'unanimité**, loin s'en faut.

Les aspects positifs de cette méthode

- Abstraction de haut niveau:
  - Modélisation des concepts métier.
  - Modélisation des règles métier
  - Modélisation des endpoints (pour une API)
  - Résultat = Meilleure vision d'ensemble du logiciel
- Facilite la communication avec les équipes non techniques
- Réutilisation des modèles dans d'autres contextes.
- Automatisation de certaines tâches comme la production de code et les tests
- Facilite le changement de technologie ou de stack technique.

Les aspects négatifs de cette méthode

- Complexité des modèles
  - Certains critiques, comme Martin Fowler, souligne la complexité de certains modèles
  - Certains modèles sont plus complexes que le code lui-même.
- Maintenabilité des modèles
  - Nécessite des compétences techniques très spécifiques
  - Des mises à jour fréquentes peuvent rendre la maintenance complexe
- Synchronisation entre modèles et code difficile si le code a été modifié directement.
- Génération de code de qualité variable

Les aspects négatifs de cette méthode

- Les développeurs ne se focalisent pas sur la valeur ajoutée pour le client mais sur des couches d'abstraction que le client n'a pas commandées.
- La courbe d'apprentissage est élevée.
  - Une nouvelle personne dans l'équipe doit se former à de nombreux outils.

Alors que penser de cette approche ?

- Il y a des points intéressants qui méritent d'être soulignés.
- Génération des endpoints d'une API à partir d'un modèle : permet d'avoir une vision d'ensemble de l'API.
- Création d'un modèle pour les endpoints avec RAML

RAML

- RAML: RESTful API Modeling Language
- Exemple: partie déclaration des types

```
# Exemple de spécification RAML pour une API de gestion de tâches
title: Gestion de Tâches API
version: 1.0
baseUri: /api

# Déclaration des types de données (entités)
# ---
types:
  Tache:
    properties:
      id: number
      title: string
      description: string
      completed: boolean
```

RAML

❑ Exemple : partie déclaration des endpoints.

```
# Déclaration des ressources (endpoints)
# de l'API
# ---
/taches:
  displayName: Tâches
  description: Ressource pour la gestion des tâches
  get:
    description: Récupérer la liste des tâches
    responses:
      200:
        body:
          application/json:
            example: |
              [
                {"id": 1, "title": "Faire les courses", "description": "Acheter du lait et du pain", "completed": false},
                {"id": 2, "title": "Réunion", "description": "Réunion avec l'équipe à 14h", "completed": false}
              ]
```

Génération de code Java à partir de RAML

- Utilisation de **RAML JAX-RS**
- Dans le pom.xml : mise en place du plugin MAVEN **com.phoenixnap.oss / springmvc-raml-plugin**
- **Problème** : le plugin n'a pas évolué depuis 2019 et génère des contrôleurs un peu dépassés du point de vue code.

DROOLS

- La DSL Drools Rule Language permet de décrire les règles métier.
- Il est basé sur un langage déclaratif

```
rule "Réduction pour les grosses commandes"
    when
        $order : Order(amount > 1000)
    then
        $order.applyDiscount(10);
end
```

- Ensuite, on peut intégrer ces règles métier dans des fichiers .drl et démarrer un moteur Drools

Prise en compte des règles drools dans une application Java

- Avec **Drools** il n'est pas nécessaire de générer du code Java
- On peut utiliser un moteur de règle Drools dans une classe de services.

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kieContainer = kieServices.getKieClasspathContainer();
KieSession kieSession = kieContainer.newKieSession();
// Création d'une personne
Person person = new Person();
person.setAge(70);
// Insertion de la personne dans la session Drools
kieSession.insert(person);
// Exécution des règles
kieSession.fireAllRules();
// Récupération des résultats
kieSession.dispose();
```

Fichier .drl

```
import com.example.Person

rule "Discount Rule"
when
    person : Person(age >= 65)
then
    person.setEligibleForDiscount(true);
end
```

MDD aujourd'hui.

- Quel intérêt d'avoir du code dans des fichiers .drl ?
  - Aucun, ce n'est pas une abstraction un modèle, mais du code.
  - En cas de refactoring, on n'aura pas d'erreurs de compilation dans ces fichiers
- Quand on creuse on voit que MDD a été un feu de paille.
- Les outils dans l'écosystème Java autour de MDD ne sont plus maintenus ou sont très peu nombreux.
- Certains disent que MDD est mort, d'autres que c'est une excellente méthodologie boudée par l'industrie I.T., etc.

TP n°5

TP de modélisation

TP n°5: réalisez le diagramme de classes et le diagramme entités relations à partir des données mises à votre disposition

FIN