MODELISATION UML

1. UML, C’EST QUOI?

1.1 Définition.(Source: Wikipédia)

Le Langage de Modélisation Unifié, de l'anglais Unified Modeling Language (UML), est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système. Il est couramment utilisé en développement logiciel et en conception orientée objet

1.2 Utilisation.(Source: Wikipédia)

UML est destiné à faciliter la conception des documents nécessaires au développement d'un logiciel orienté objet, comme standard de modélisation de l'architecture logicielle. Les différents éléments représentables sont :

- Activité d'un objet/logiciel
- Acteurs
- Processus
- Schéma de base de données
- Composants logiciels
- Réutilisation de composants.

Il est également possible de générer automatiquement tout ou partie du code, par exemple en langage Java, à partir des documents réalisés.

1.3 A retenir

UML se compose :

- d’éléments de modélisation pour représenter les objets
- d’un ensemble de diagramme pour en représenter le comportement

Il est capable de générer du code à partir de ces descriptions et de ses diagrammes.

UML est un langage standard pour modéliser les objets.

1.4 Quelques notions

Notion d’objet:

La programmation orientée objet consiste à modéliser informatiquement un ensemble d'éléments d'une partie du monde réel (que l'on appelle domaine) en un ensemble d'entités informatiques. Ces entités informatiques sont appelées objets. Il s'agit de données informatiques regroupant les principales caractéristiques des éléments du monde réel (taille, couleur, ...).

Un objet est caractérisé par plusieurs notions :

- **Les attributs**: Il s'agit des données caractérisant l'objet. Ce sont des variables stockant des informations d'état de l'objet

- **Les méthodes (appelées parfois fonctions membres):** Les méthodes d'un objet caractérisent son comportement, c'est-à-dire l'ensemble des actions (appelées opérations) que l'objet est à même de réaliser. Ces opérations permettent de faire réagir l'objet aux sollicitations extérieures (ou d'agir sur les autres objets). De plus, les opérations sont étroitement liées aux attributs, car leurs actions peuvent dépendre des valeurs des attributs, ou bien les modifier

- **L'identité:** L'objet possède une identité, qui permet de le distinguer des autres objets, indépendamment de son état. On construit généralement cette identité grâce à un identifiant découlant naturellement du problème (par exemple un produit pourra être repéré par un code, une voiture par un numéro de série, etc.)

=> un objet a donc une identité, un état (les attributs) et un comportement (les méthodes)

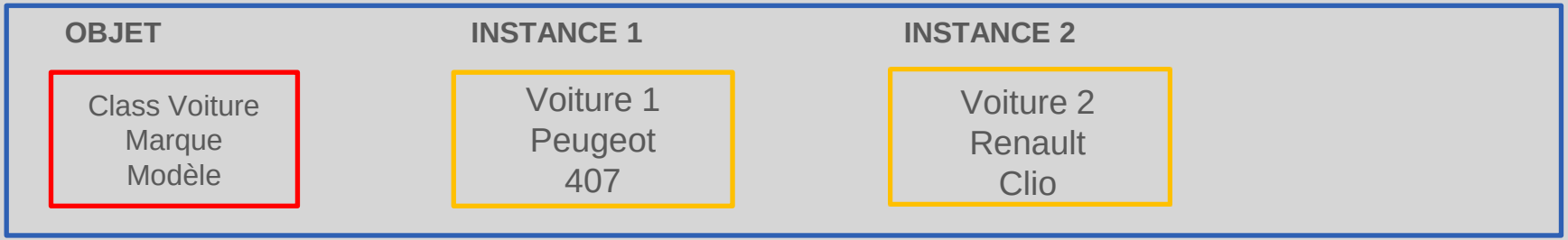
Notions de classe (ou instance)

On appelle classe la structure d'un objet, c'est-à-dire la déclaration de l'ensemble des entités qui composeront un objet. Un objet est donc « issu » d'une classe, c'est le produit qui sort d'un moule. En réalité on dit qu'un objet est une instanciation d'une classe, c'est la raison pour laquelle on pourra parler indifféremment d'objet ou d'instance (éventuellement d'occurrence).

Une classe est composée de deux parties :

- Les attributs (parfois appelés données membres) : il s'agit des données représentant l'état de l'objet
- Les méthodes (parfois appelées fonctions membres): il s'agit des opérations applicables aux objets

Si on définit la classe voiture, les objets Peugeot 208, Renault Clio seront des instanciations de cette classe. Il pourra éventuellement exister plusieurs objets Peugeot 208, différenciés par leur numéro de série.
Mieux: deux instanciations de classes pourront avoir tous leurs attributs égaux sans pour autant être un seul et même objet. C'est le cas dans le monde réel, deux T-shirts peuvent être strictement identiques et pourtant ils sont distincts. D'ailleurs, en les mélangeant, il serait impossible de les distinguer...



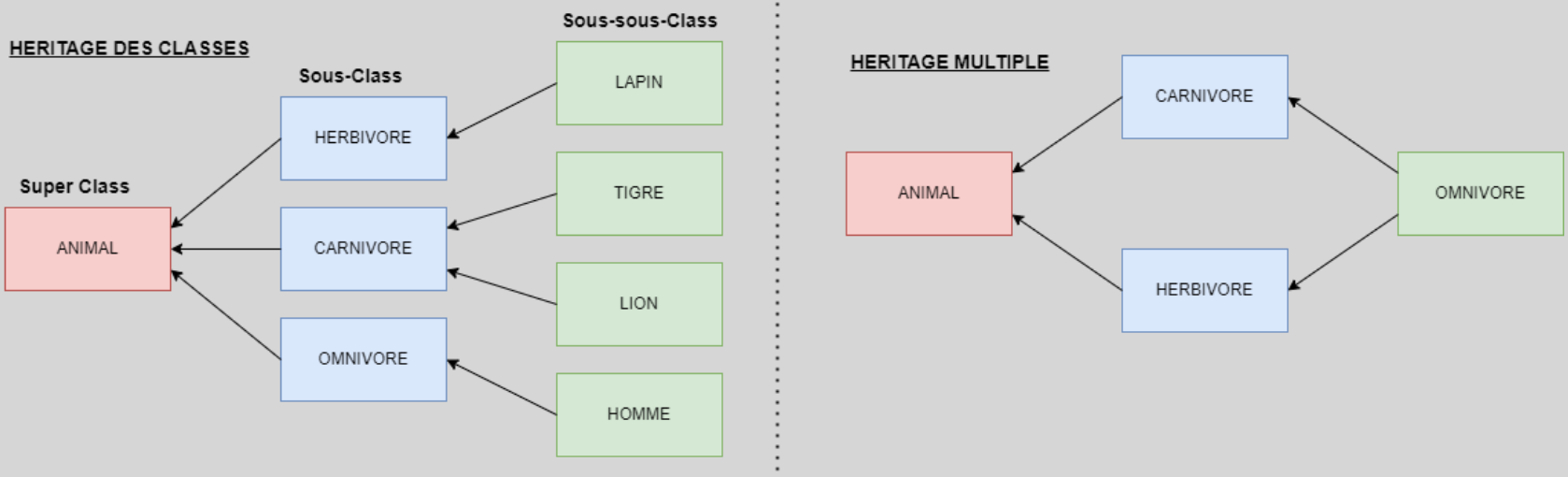
L’encapsulation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés.
L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.
Aussi, cela permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent les droits d'accès aux données selon que l'on y accède par une méthode de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque. Il existe trois niveaux de visibilité:

- **publique**: les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité public. Il s'agit du plus bas niveau de protection des données
- **protégée**: l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe ainsi que des classes dérivées
- **privée**: l'accès aux données est limité aux méthodes de la classe elle-même. Il s'agit du niveau de protection des données le plus élevé

L’héritage

L'héritage (en anglais inheritance) est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante.
Le nom d'héritage (pouvant parfois être appelé dérivation de classe) provient du fait que la classe dérivée (la classe nouvellement créée) contient les attributs et les méthodes de sa superclasse (la classe dont elle dérive). L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.
Par ce moyen on crée une hiérarchie de classes de plus en plus spécialisées. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir (on comprend encore un peu mieux l'intérêt pour l'entreprise qui vend les classes de protéger les données membres grâce à l'encapsulation...).



L’acteur (en UML)

Un acteur, au sens UML, représente le rôle d'une entité externe (utilisateur humain ou non) interagissant avec le système. Il est représenté par un bonhomme en fil de fer (en anglais stick man).

On représente généralement à gauche l'acteur principal (idéalement il y en a un seul), et à droite les acteurs secondaires. Il est à noter qu'un utilisateur peut amené à jouer plusieurs rôles vis-à-vis du système et à ce titre être modélisé par plusieurs acteurs.

Processus (en UML)

C’est l’ensemble des activités effectuées par les acteurs afin de répondre à un type d'événement.
Une activité est un ensemble de tâches ou actions exécutés par des machines ou des êtres humains. Elles peuvent être déclenché par un événement connu et définit par le système d’information (ex: sauvegarde bdd)

1.5 Les diagrammes UML

Il existe 14 diagrammes différents qui décrivent chacun un point de vue particulier du système.

Ces différents diagrammes peuvent être regroupés dans 2 modes de représentations:

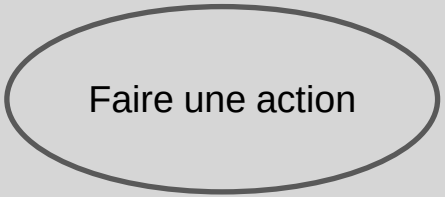
- Un mode statique (ou structurel)
- Un mode dynamique (ou fonctionnel)

Chaque diagramme représente à sa manière graphiquement notre système.

Le diagramme d'usage

C'est un moyen simple d'exprimer les besoins, de recueillir, d'organiser et de recenser les fonctionnalités d'un système. Il représente les acteurs et les cas d'utilisations. Les acteurs peuvent être des humains, des machines ou des organisations.

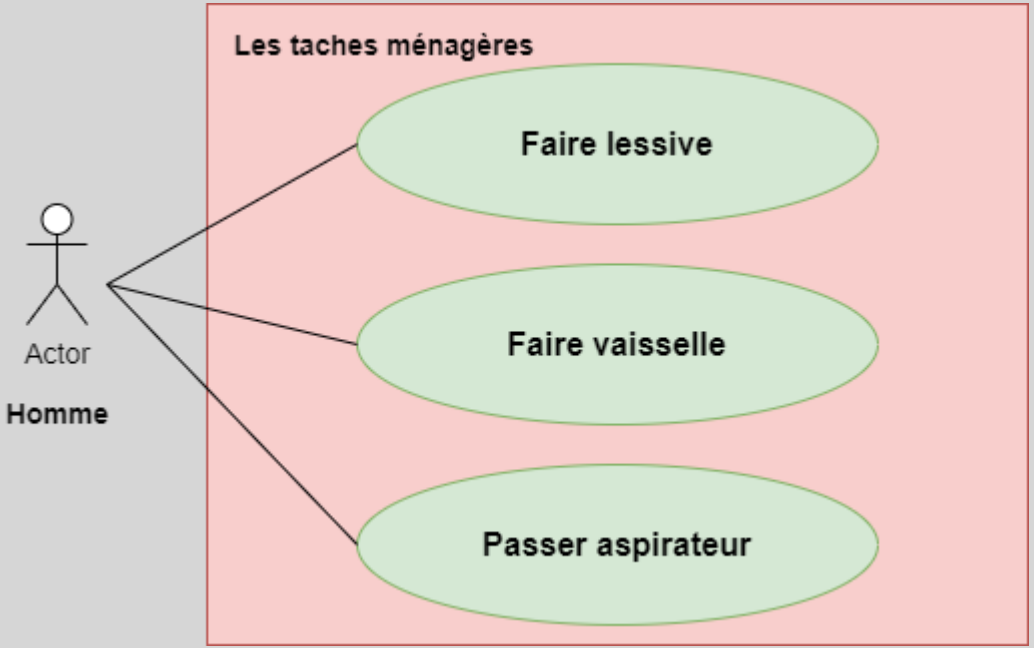
Un cas d'utilisation



1.6 Diagramme d’usage

exemple diagramme d’usage

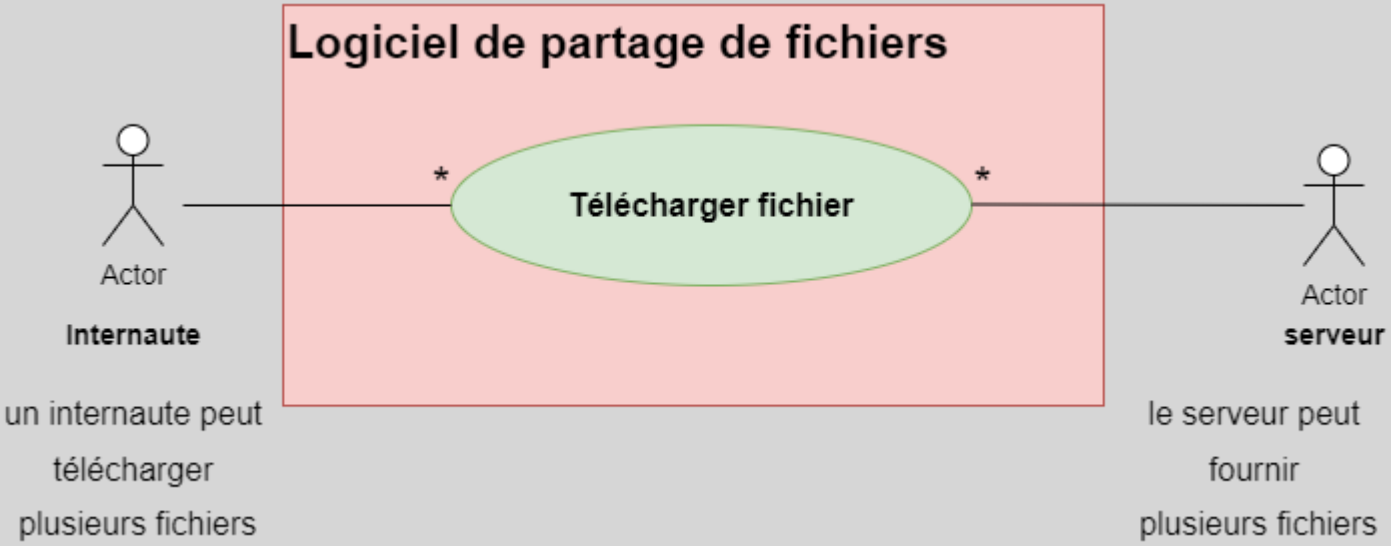
On représente la frontière du système par un cadre dans lequel on retrouve les cas d’utilisation et le nom de celui-ci. Les acteurs sont à l’extérieur du cadre. Le trait entre acteur et usage est appelé relation.



La multiplicité

La multiplicité d'une relation exprime le nombre de relation possible

- * => plusieurs (ex: une boutique peut avoir plusieurs produits (sans limite))
- n => exactement n (ex: un plateau d'échiquier contient exactement 64 cases)
- n..m => entre n et m (ex: pour un compte Netflix il peut y avoir entre 1 et 4 utilisateurs)

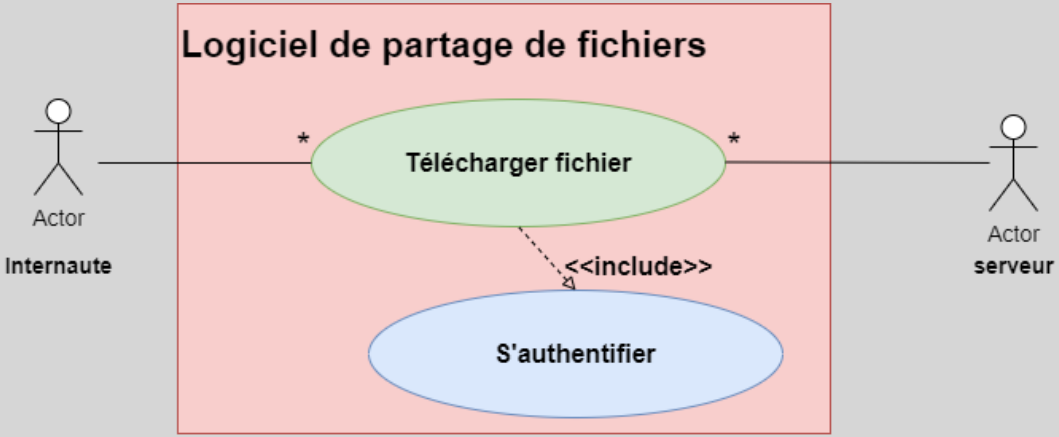


Les relations d’extensions et inclusions

Pour représenter des relations complexes entre différents cas d'utilisation, nous pouvons utiliser les relations d'extensions et d'inclusions.

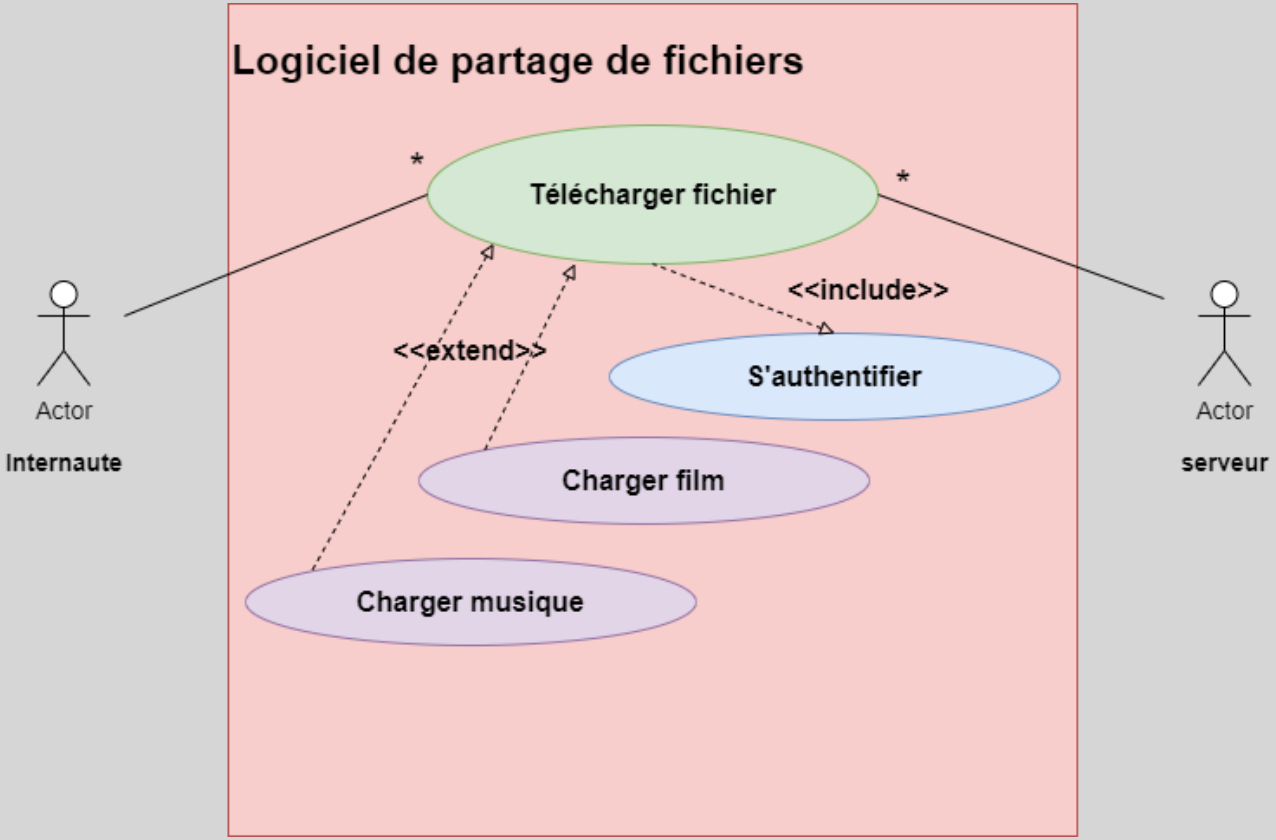
Étendre la relation (extend) : le cas d'utilisation est **facultatif** et vient après le cas d'utilisation de base. Il est représenté par une flèche en pointillé dans la direction du cas d'utilisation de base avec la notation <<extend>>.

Inclure la relation (include) : le cas d'utilisation est **obligatoire** et fait partie du cas d'utilisation de base. Il est représenté par une flèche en pointillé dans la direction du cas d'utilisation inclus avec la notation <<include>>.

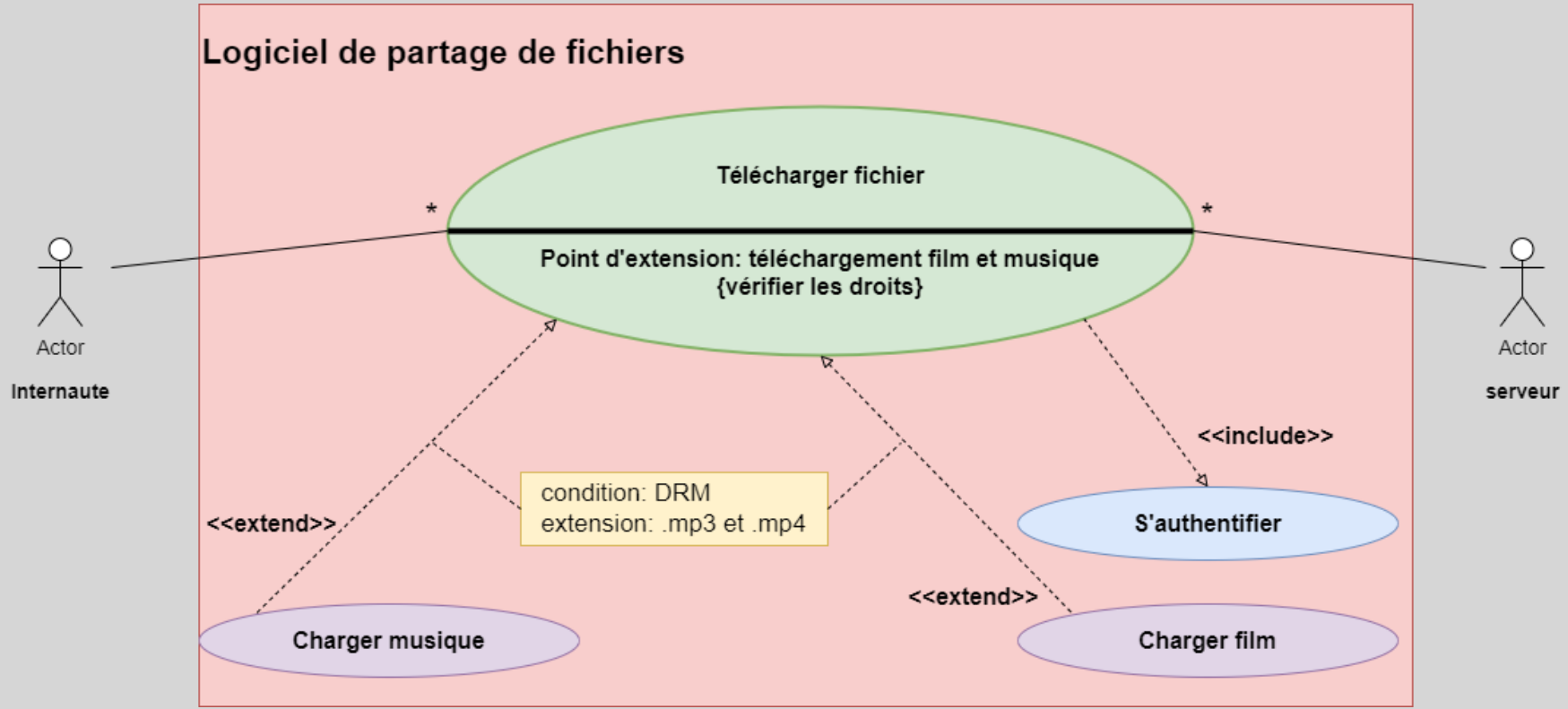


L'internaute doit obligatoirement s'authentifier pour pouvoir télécharger le fichier

On peut aller plus loin en étendant la relation

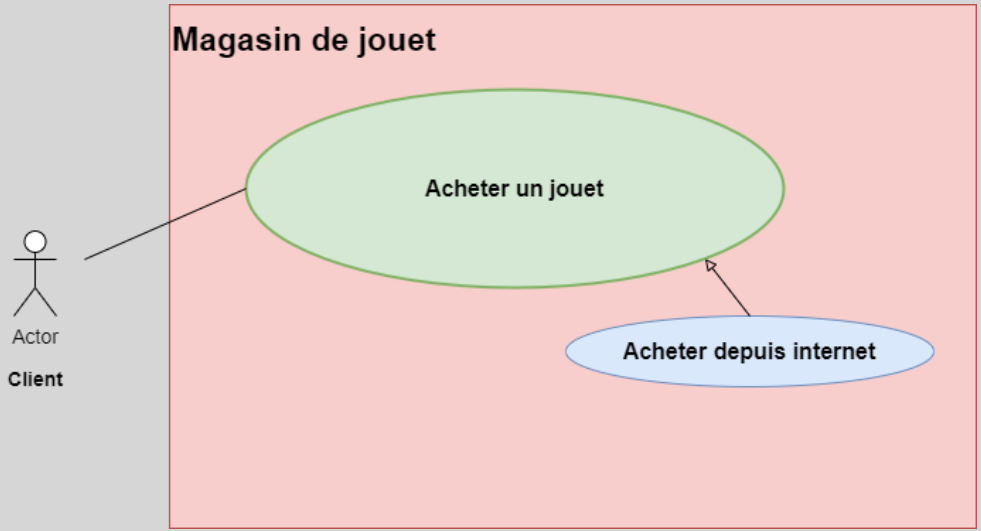


Pour aller encore plus loin, on ajoute des conditions sur les relation d’extensions



La relation de généralisation (ou spécialisation)

Elle permet de donner une indication à la relation, elle est représenté par une flèche pleine



Savoir identifier les acteurs

- Les acteurs sont des entités externes au système et interagissent avec lui. Il est utile de les identifier pour bien cerner l'interface que le système va devoir avoir.
- Au delà des acteurs, le système peut aussi interagir avec d'autres systèmes qui deviennent de ce fait acteur.
- Les acteurs doivent être nommés selon leur rôle.

Savoir identifier les cas d'utilisations

- Les cas d'utilisations doivent répondre aux exigences fonctionnelles du système.
- Chaque cas est une fonction métier selon le point de vue des acteurs.
- Donc pour identifier les cas d'utilisation, il faut comprendre pourquoi et comment un acteur se sert du système.
- Il faut éviter les redondances et limiter le nombre de cas.
- On nomme les cas avec un verbe à l'infinitif suivi d'un complément en se plaçant du point de vue de l'acteur.

Pour bien cerner un cas d'utilisation, on utilisera la description textuelle suivante :

Nom : utiliser une tournure à l'infinitif (ex: faire la lessive).
Objectif : une description résumée pour bien comprendre le cas d'utilisation.
Acteur principaux : ceux qui vont réaliser le cas d'utilisation.
Acteur Secondaire : ceux qui reçoivent une information à l'issue de la réalisation du cas d'utilisation.

Pour bien cerner un cas d'utilisation, on utilisera la description textuelle suivante :

Nom : utiliser une tournure à l'infinitif (ex: faire la lessive).

Objectif : une description résumée pour bien comprendre le cas d'utilisation.

Acteur principaux : ceux qui vont réaliser le cas d'utilisation.

Acteur Secondaire : ceux qui reçoivent une information à l'issue de la réalisation du cas d'utilisation.

Les préconditions : description de l'état nécessaire du système à la possible réalisation du cas.

Les scénarios : C'est une description des échanges entre l'acteur et le cas d'utilisation, ils sont de trois types : *nominal* (tous ce passe bien), *alternatif* (variante du cas nominal) et *exception* (décrivant le cas d'erreur).

Les postconditions : description de l'état du système après réalisation des différents scénarios.

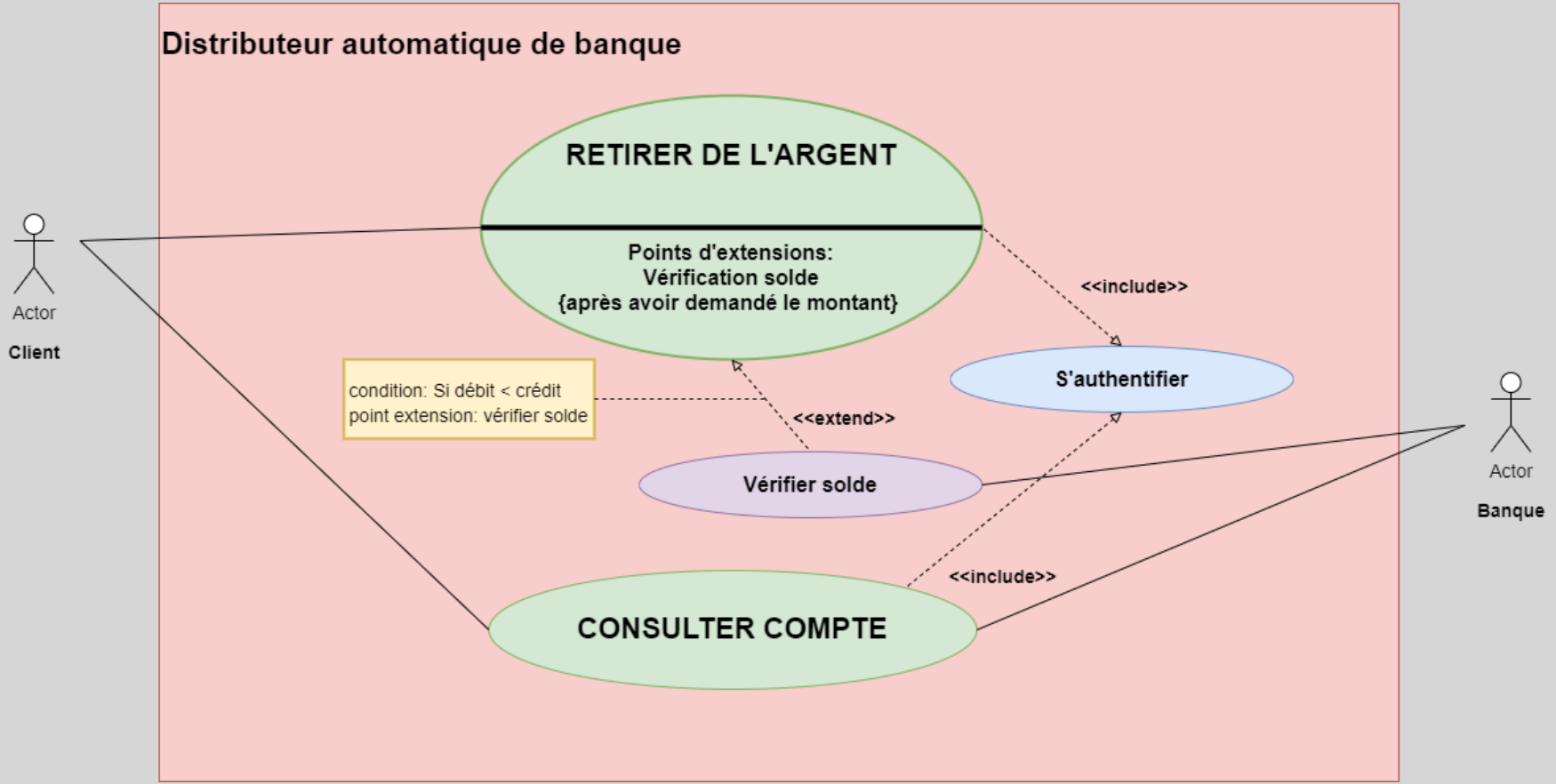
On peut compléter ces informations avec des spécifications non fonctionnelles (contrainte technique) et une description des besoins en termes d'interface.

Exercice sur le diagramme d’usage

Réaliser le diagramme d’usage de l'utilisation d’une borne interactive d’une banque pour des opérations classique (retrait, consultation).

⇒ Bien identifier les acteurs
⇒ Analyser les actions à effectuer
⇒ Prendre en considération les préconditions

Correction diagramme d’usage



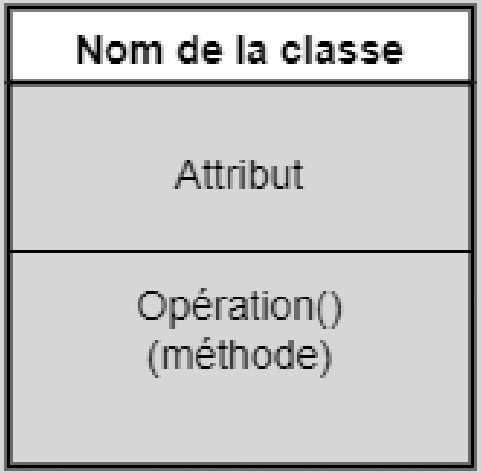
1.7 Diagramme de classes

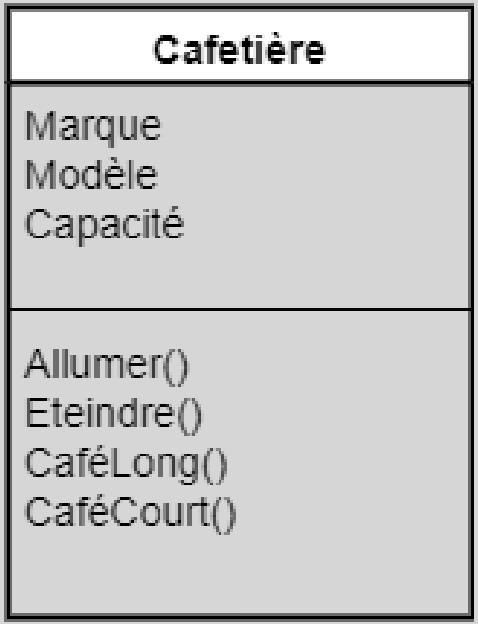
C’est le diagramme de la structure interne du système.
Il fournit une représentation abstraite des objets du système qui vont interagir ensemble.
Il permet de modéliser les classes et leurs relations indépendamment d’un langage de programmation.
Il est obligatoire dans le cadre de la POO.
Ce diagramme est composé de classe.

Pour mémoire : une classe est la description formelle d’un ensemble d’objets ayant une sémantique et des caractéristiques communes.
Un objet est une instance d’une classe : c’est une entité dotée d’une identité, d’un état et d’un comportement.
Exemple : si on considère que voiture est un concept abstrait (classe), on peut dire que Clio en est une instance (objet).
La classe est composée de caractéristiques permettant de spécifier l’état et le comportement de ses instances.

Les caractéristiques d’un objet sont des attributs et des opérations et des terminaisons d’association.
Etat d’un objet : ce sont les propriété structurelles qui réunissent les attributs et les terminaisons d’associations. Les attributs sont utilisés pour les données, les terminaisons réfères l’identité des classes reliées.
Les attributs prennent des valeurs lors de l'instanciation de la classe.
Comportement d’un objet : les opérations décrivent les comportements que l’on peut invoquer.
Ce sont des fonctions (méthodes) qui récupèrent ou pas les valeurs des attributs de l’objet pour les modifier et/ou produire un résultat.

Représentation graphique de base

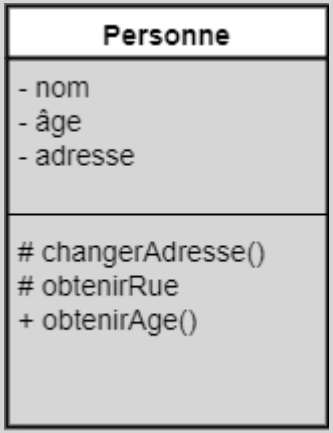




Visibilité des attributs et/ou des méthodes

- public => +
- Privé => -
- Protected => #
- Package => ~

protected : accessible dans la classe ou dans une classe enfant ou dans une classe du même package. l'encapsulation veut que les attributs soient private ou protected.

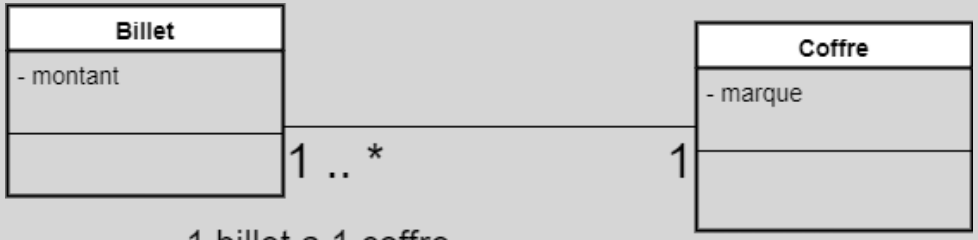


Relation entre 2 classes

L'association est le lien entre 2 classes (binaire) ou plusieurs (n-aire). elle peut posséder un nom, une cardinalité, des rôles, des terminaisons.

Les cardinalités

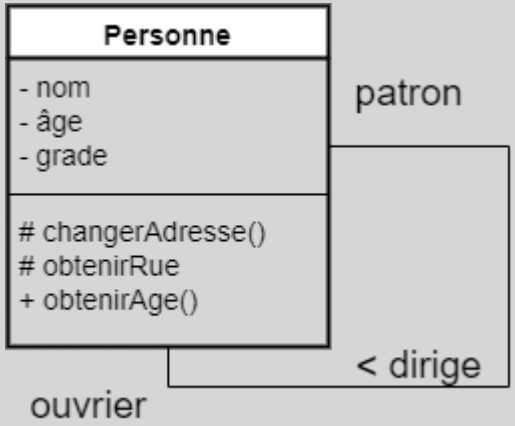
- 1..1 / 1 => exactement 1
- * / 0..* => plusieurs
- 1..* => au moins 1
- 1..n => de 1 à n



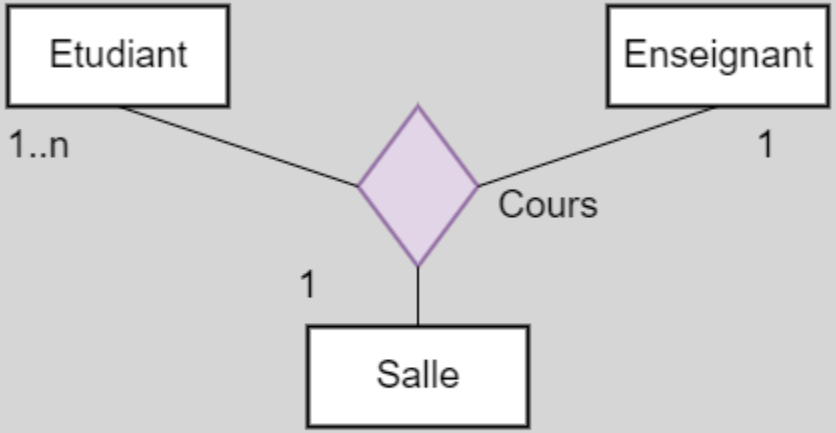
1 billet a 1 coffre
1 coffre a 1 ou plusieurs billets

Association réflexive

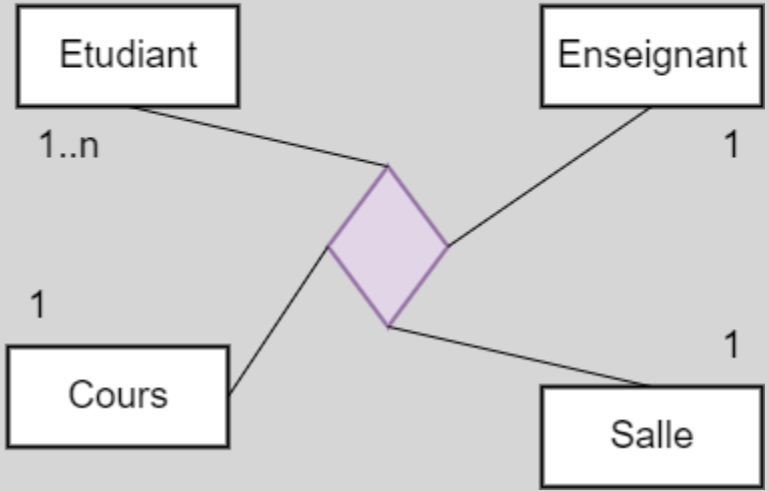
La relation réflexive marque la relation d'une classe sur elle même.
Relies les instances différentes d'une même classe



Association n-aire

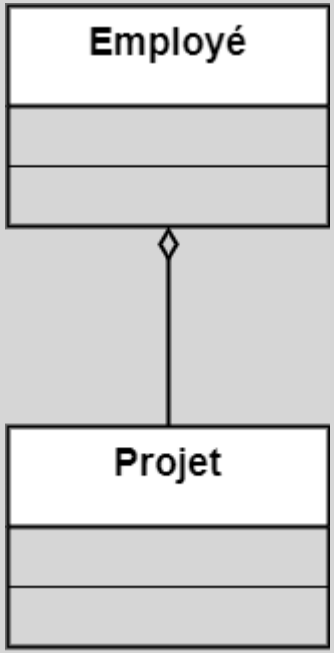


Quand une association doit avoir des propriétés on fait une classe d'association



Association spécialisée

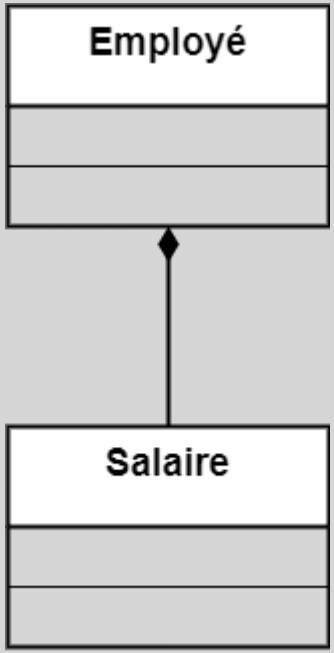
Agrégation : La relation d’agrégation exprime une relation entre deux classes où la « classe A » a un « objet B » sans que « l’objet B » n’est besoin de « l’objet A »



Un employé a des projets
Mais les projets peuvent exister sans employé

Composition

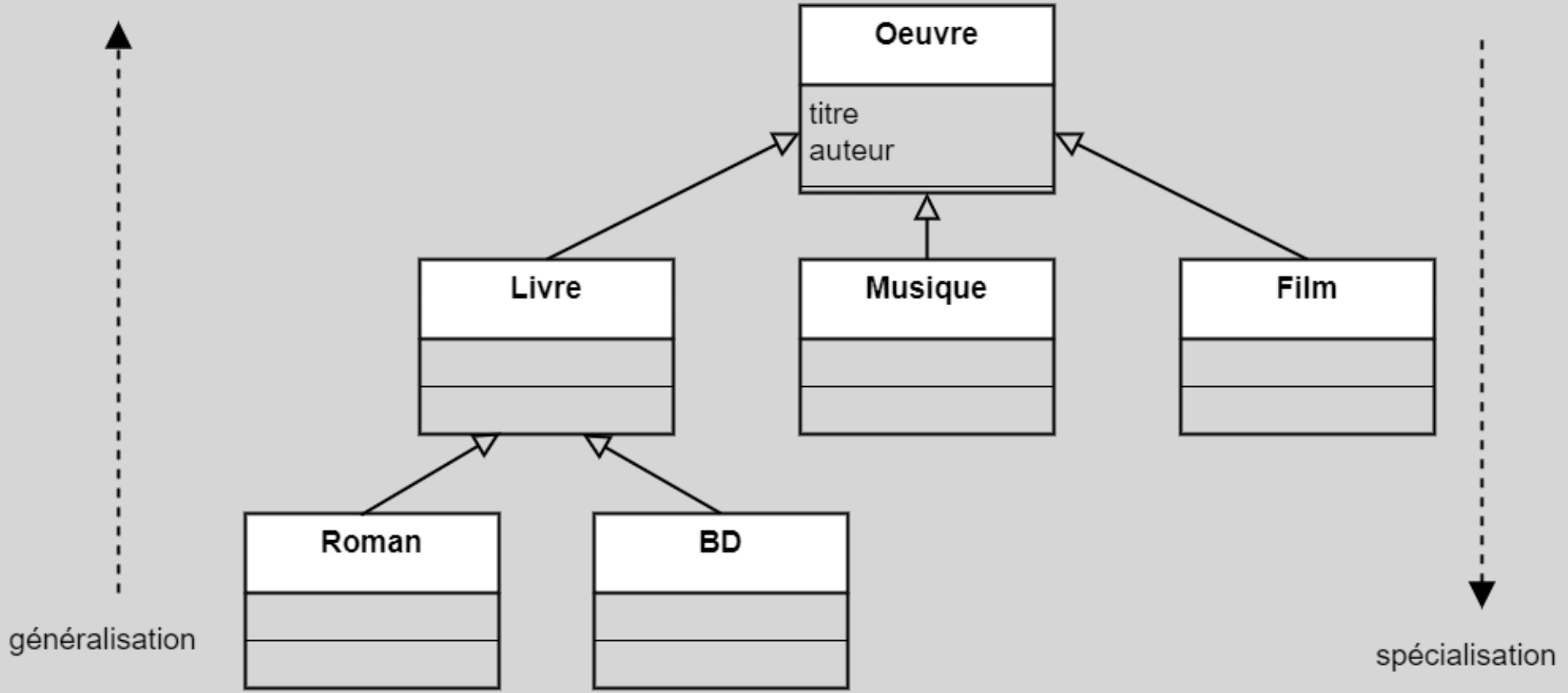
la relation de composition caractérise une relation entre deux classes de type “fait partie de” : on est sure que si la classe B fait partie de la classe A, si l’objet A cesse d'exister, B cesse d’exister.



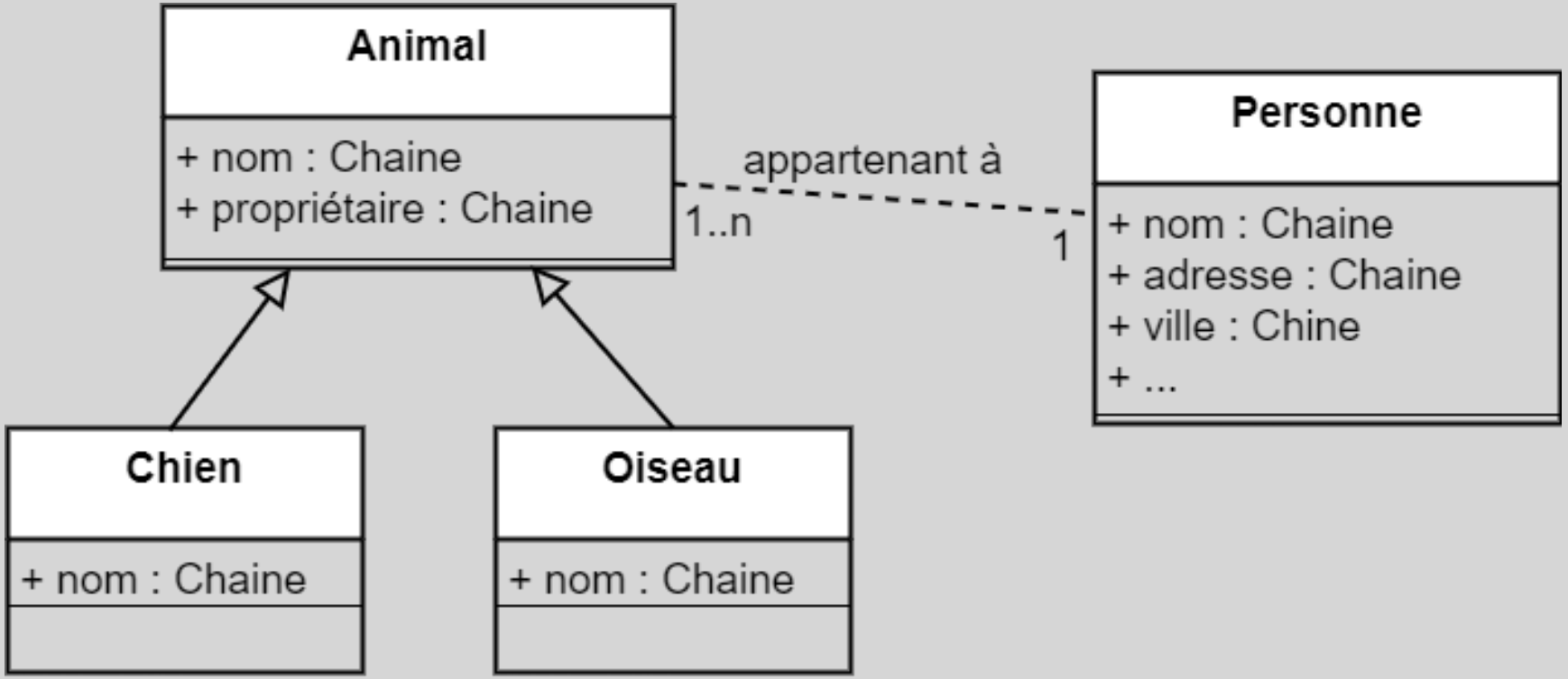
L’instance d’un salaire fait partie de l’instance de l’employé
Si on supprime l’employé, son salaire est aussi supprimé

Héritage

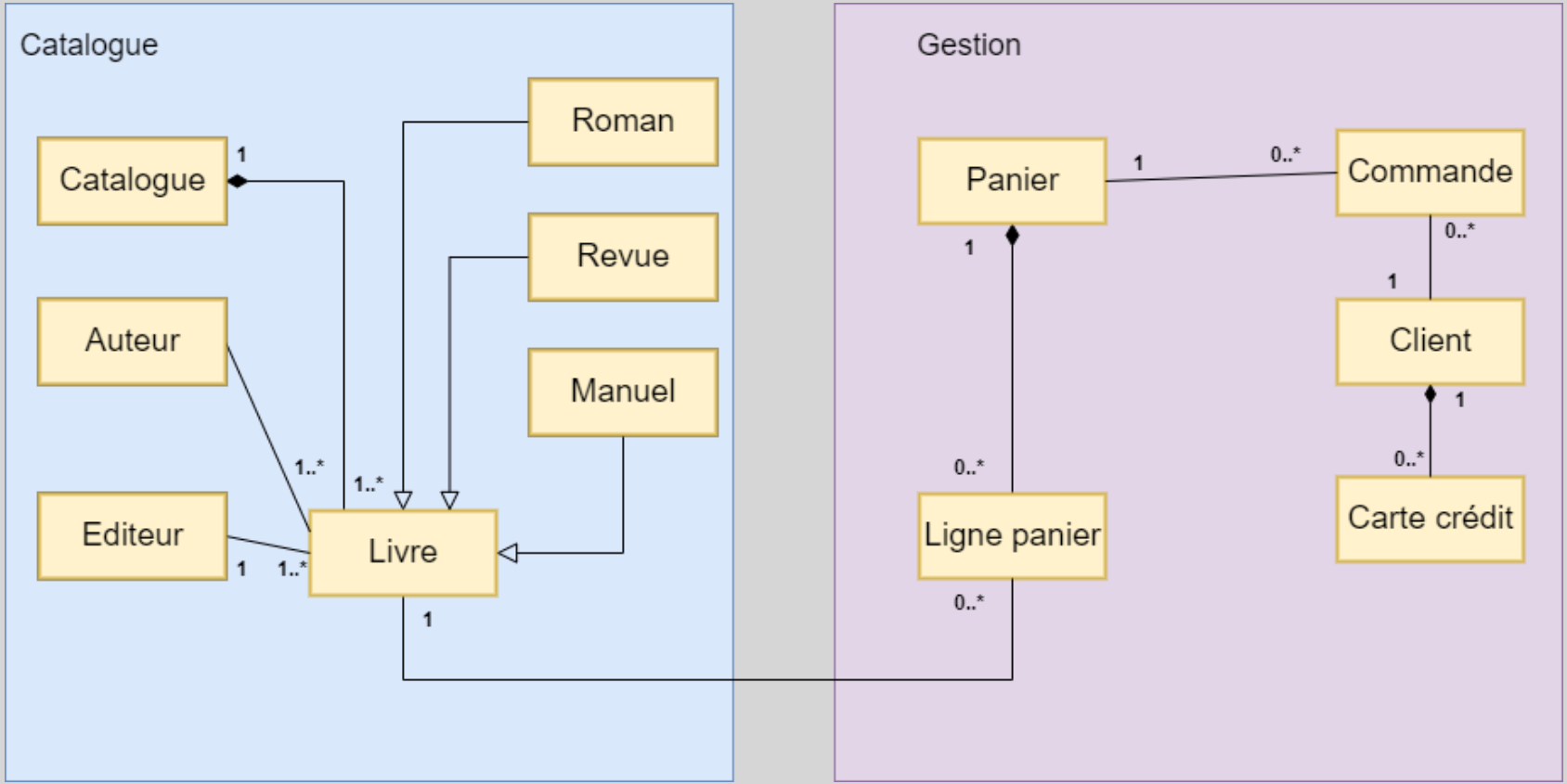
Plus on descend dans « l'arbre » plus on se spécialise, plus on remonte, plus on se généralise



Complexité du système: exemple simple



Complexité du système: exemple plus complexe avec organisation en package



Etude de cas: une bibliothèque

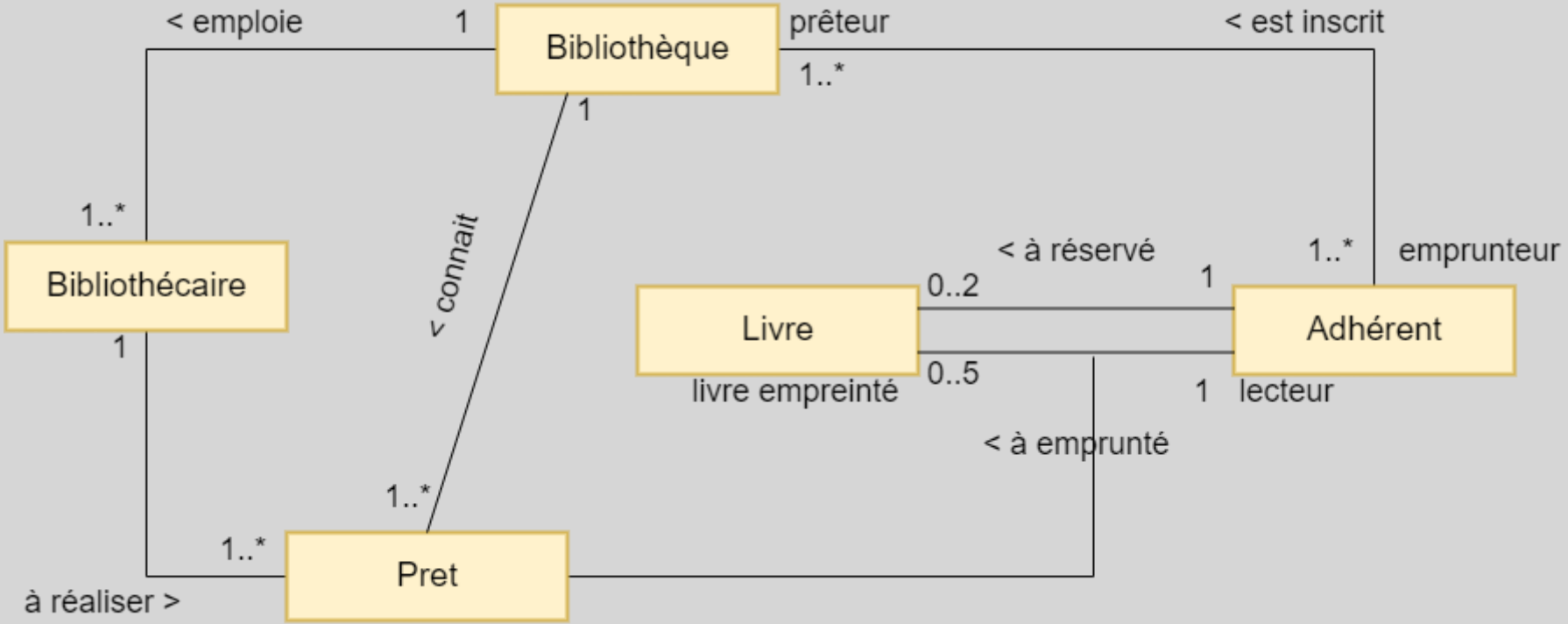
Un gérant de bibliothèque désire automatiser la gestion des prêts.
Il commande un logiciel permettant aux utilisateurs de connaître les livres présents, d'en réserver jusqu'à un certain nombre.

- L'adhérent peut connaître la liste des livres qu'il a empruntés ou réservés.
- L'adhérent possède un mot de passe qui lui est donné à son inscription.
- L'emprunt est toujours réalisé par les employés qui travaillent à la bibliothèque. Après avoir identifié l'emprunteur, ils savent si le prêt est possible (nombre max de prêts = 5), et s'il a la priorité (en réservant le livre max=2).
- Ce sont les employés qui mettent en bibliothèque les livres rendus et les nouveaux livres. Il leur est possible de connaître l'ensemble des prêts réalisés dans la bibliothèque

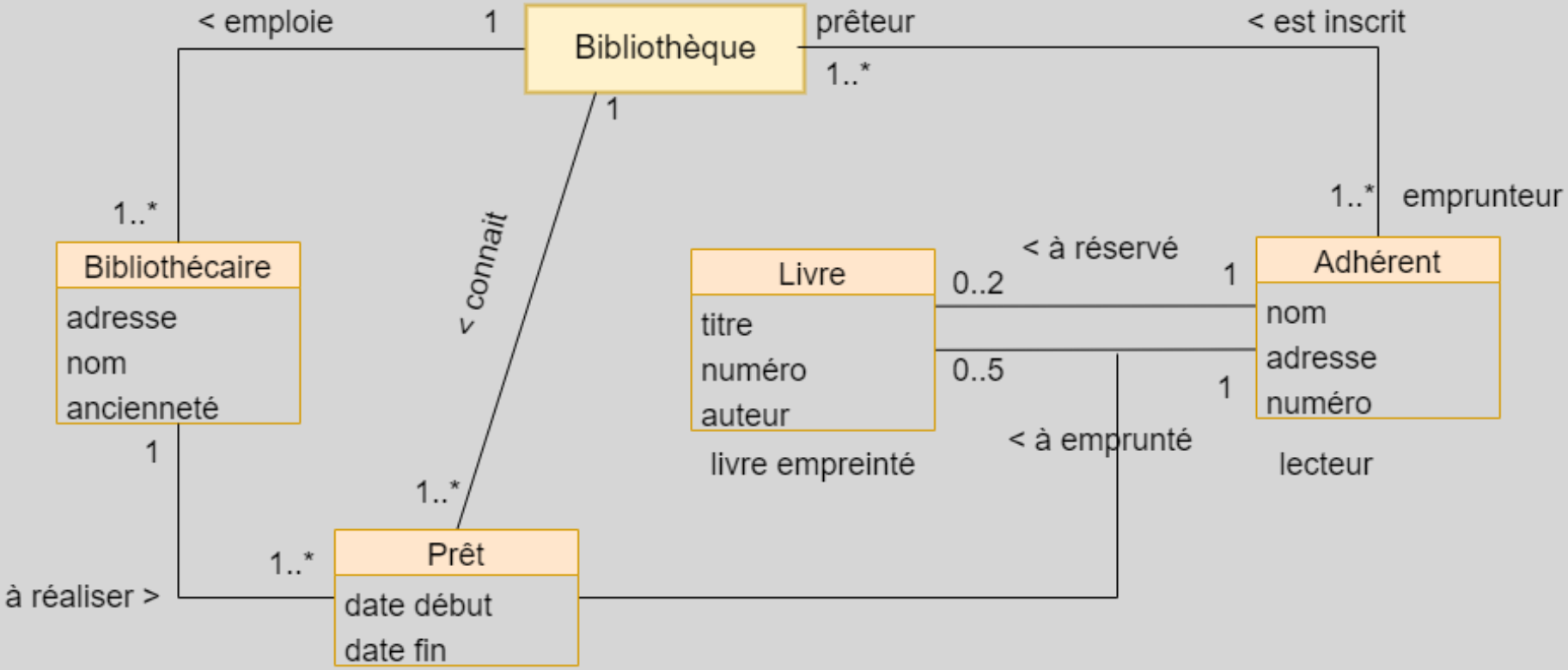
On identifie d'abords les classes :

Bibliothèque : Organisme gérant les prêts de livre à des adhérents et géré par ses employés
Employé : personne travaillant dans la bibliothèque
Adhérent : personne inscrite à la bibliothèque
Livre : ouvrage ouvert au prêt
Prêt : un prêt répertorie un livre, une durée et une date de prêt ainsi qu'un adhérent

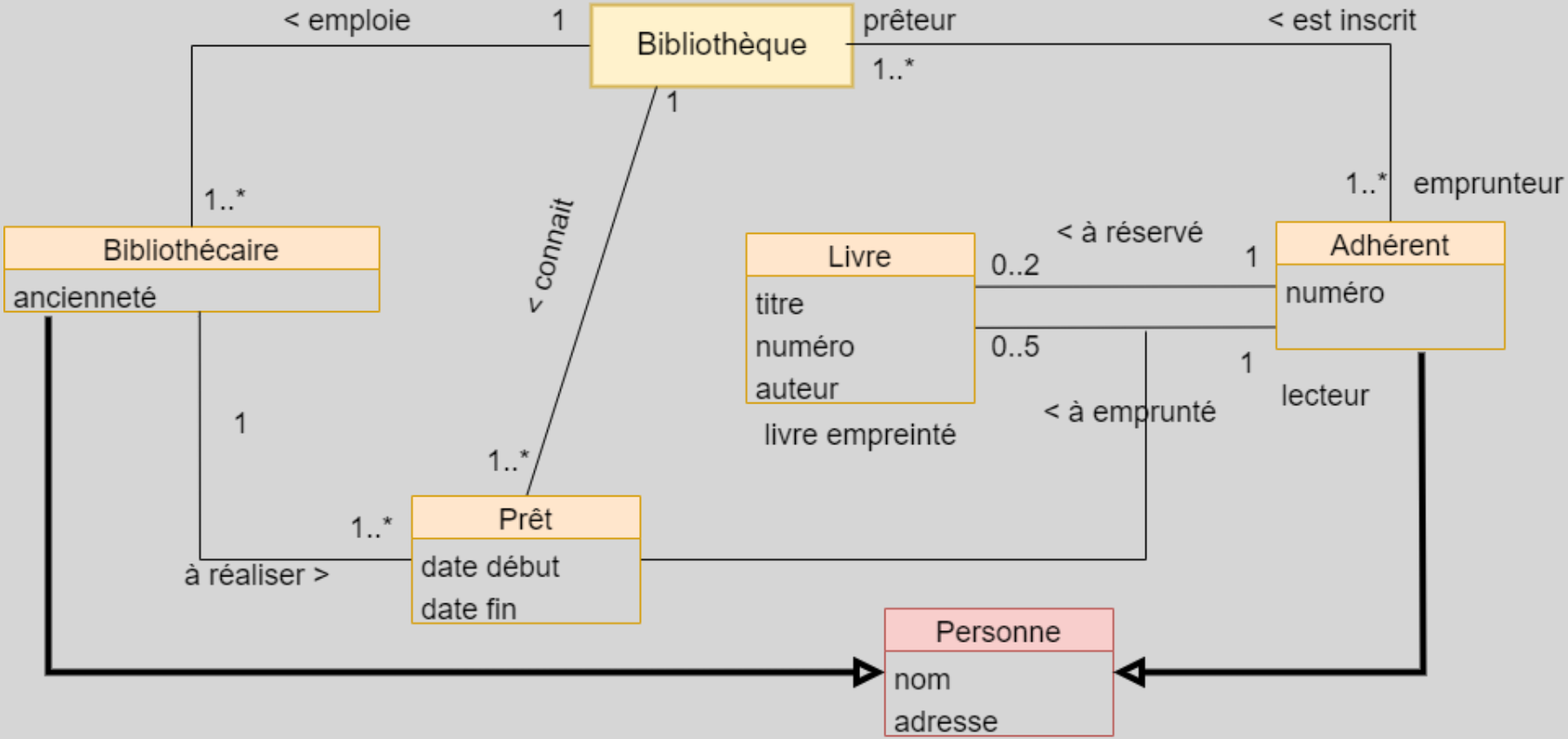
On pose les classes et on cherche les associations



On cherche les attributs



On cherche les héritages



exercice:

Achat de voiture

Une personne physique peut avoir jusqu'à trois sociétés (personnes morales) qui l'emploient.

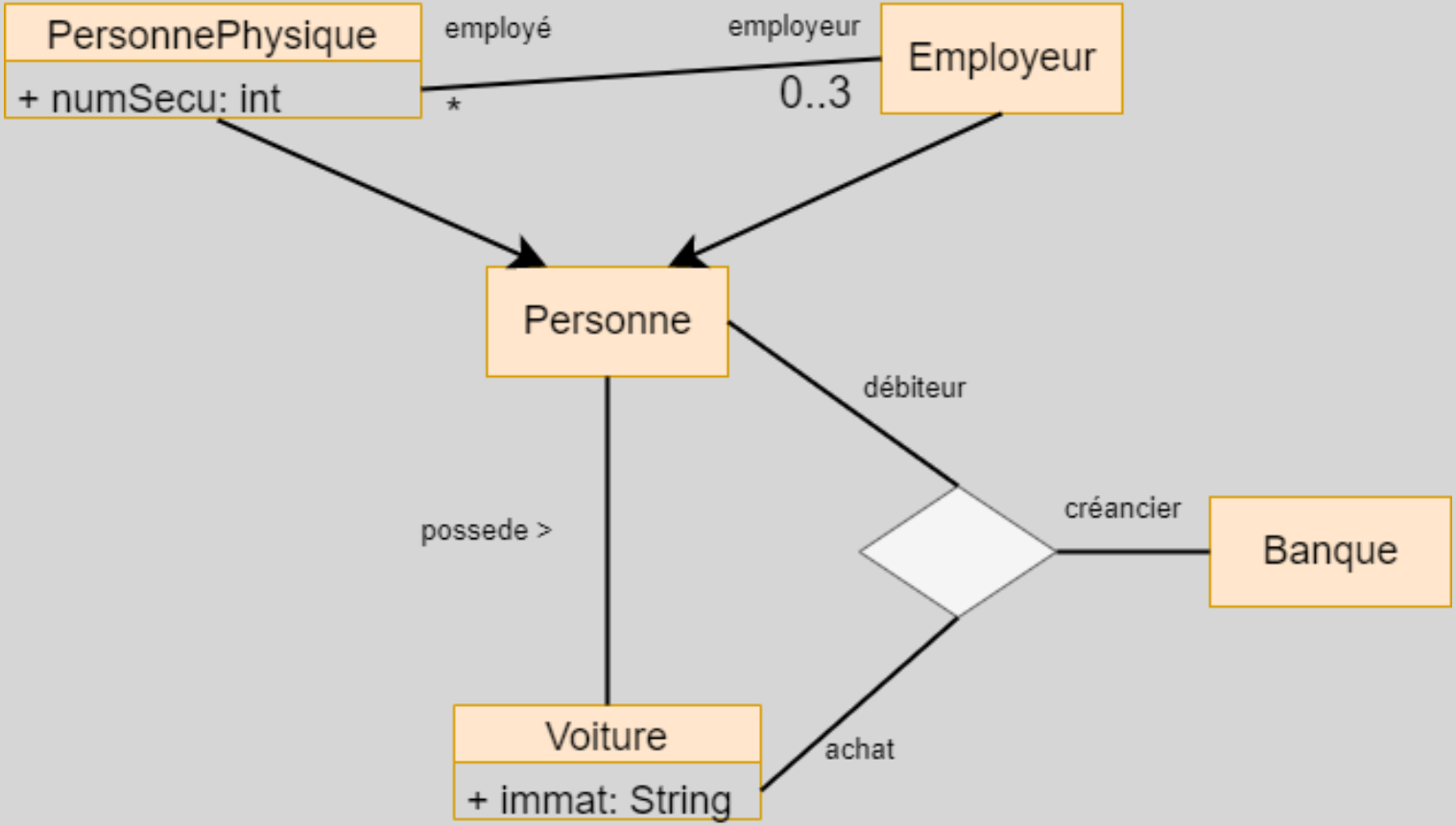
Chaque personne physique possède un numéro de sécurité sociale qui l'identifie.

Une voiture a un numéro d'immatriculation.

Une voiture est la propriété d'une personne (physique ou morale).

Un emprunt dans une banque peut être demandé pour l'achat d'une voiture.

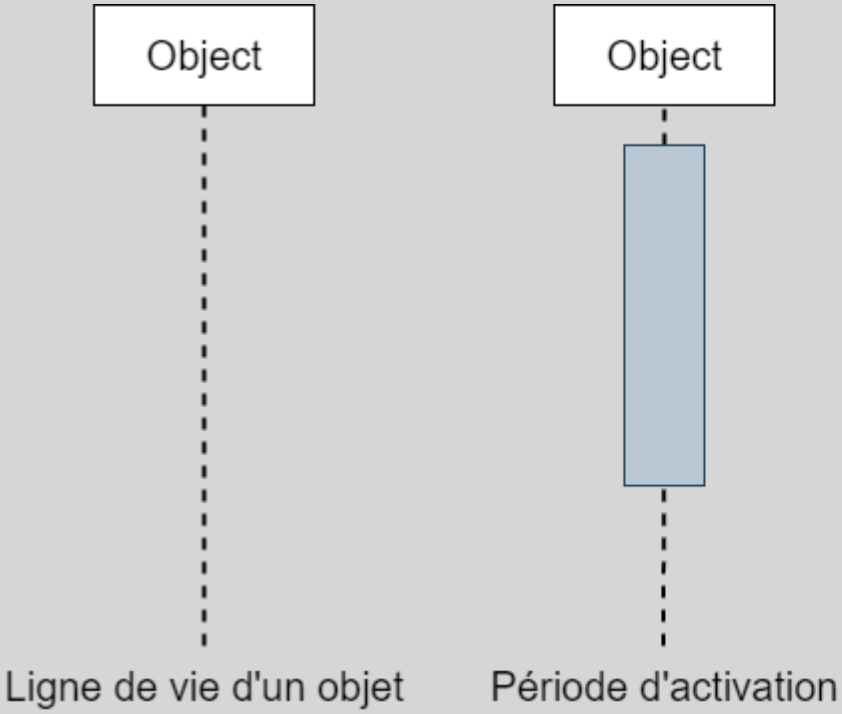
Correction exercice:

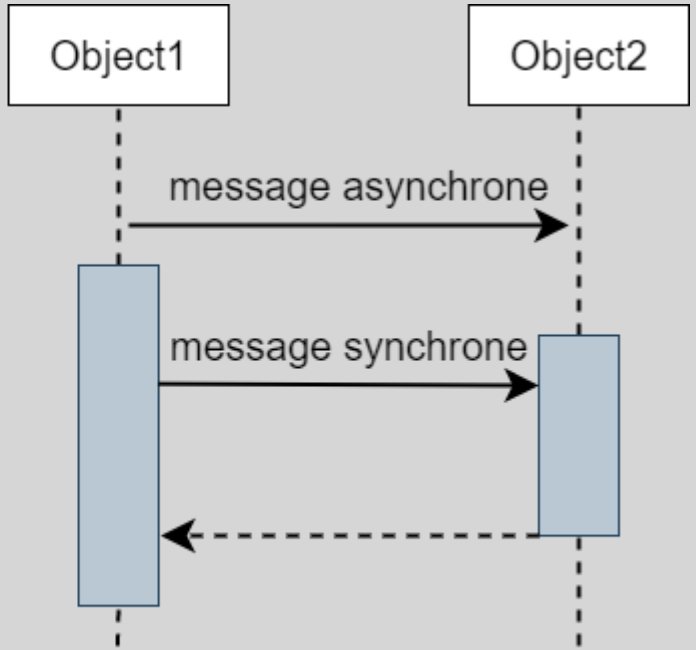


1.8 Diagramme de séquence

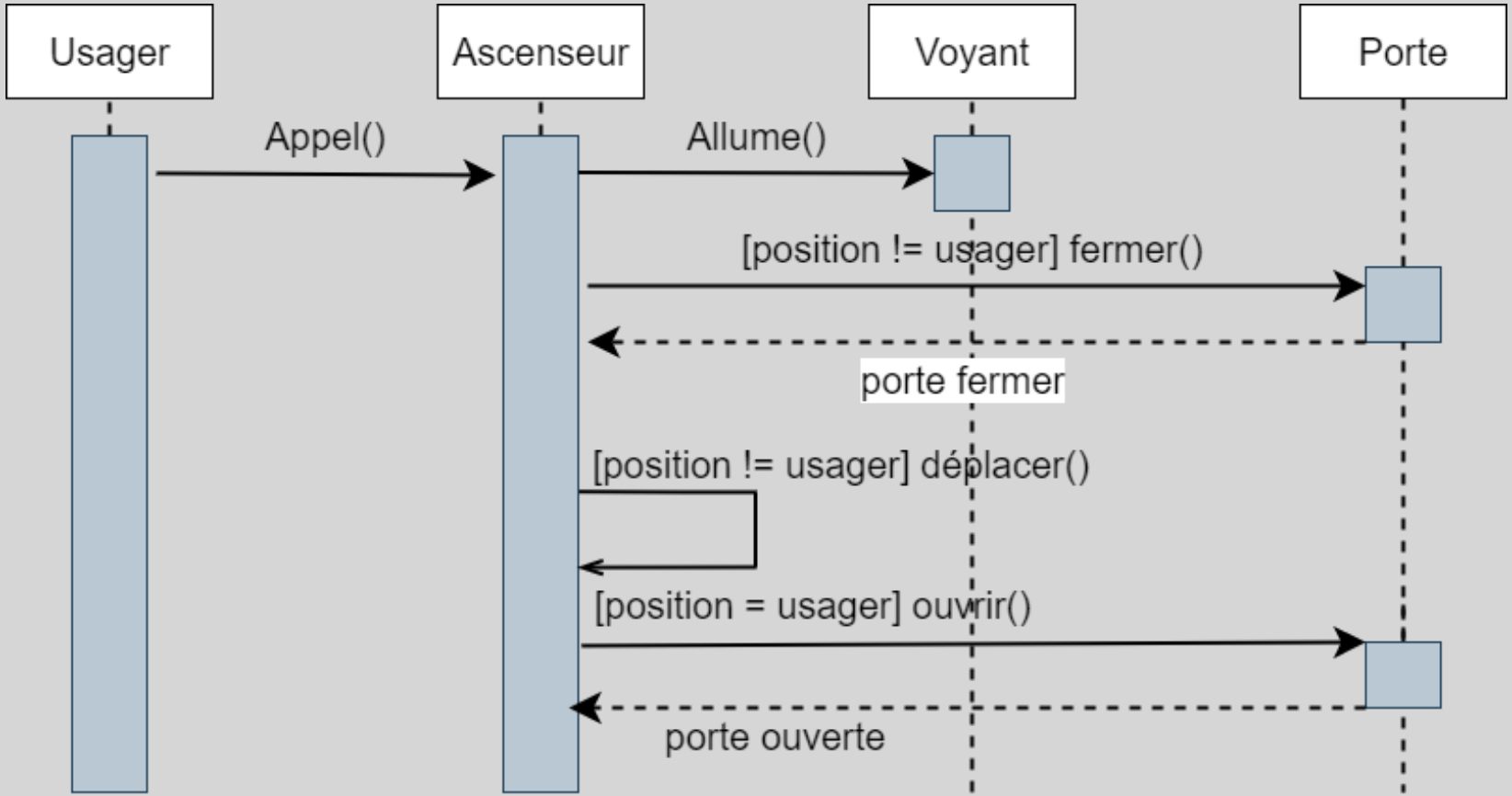
Le diagramme de séquence permet de représenter graphiquement l'ordonnancement du dialogue entre les objets :

- il permet la visualisation de l'aspect temporel des actions
- il permet de connaître le sens des interactions





Exemple appel d’un ascenseur



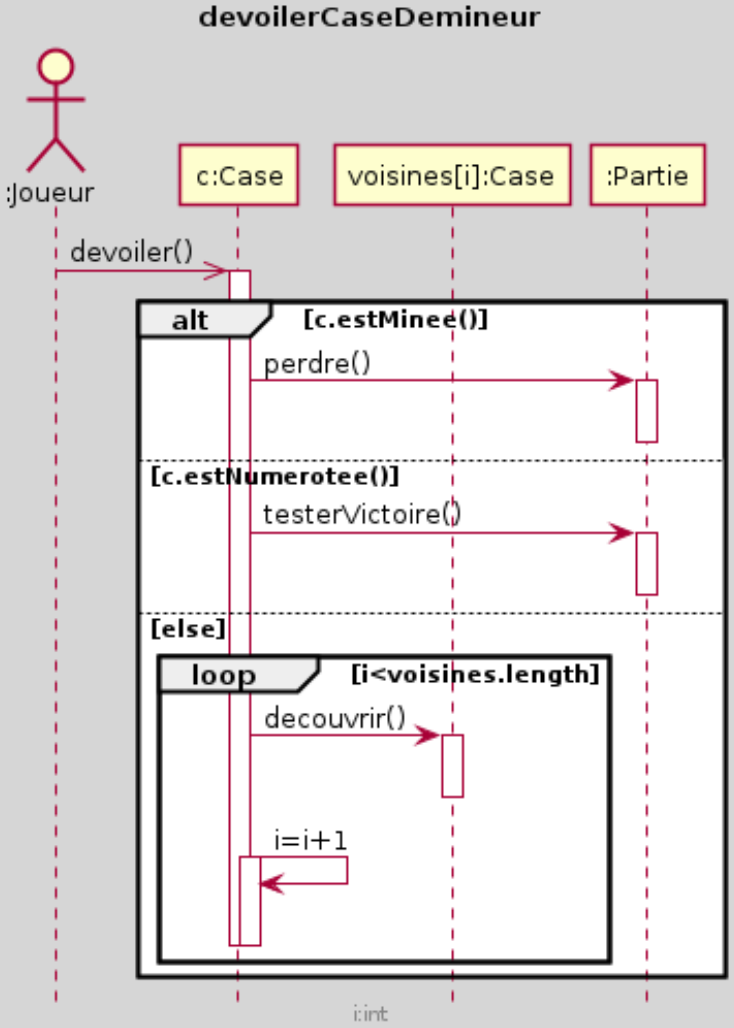
Exemple: dévoiler case démineur

Opérateurs de flux de contrôle

- **opt** (facultatif*) : Contient une séquence qui peut ou non se produire. Dans la protection, vous pouvez spécifier la condition sous laquelle elle se produit.
- **alt** : Contient une liste des fragments dans lesquels se trouvent d'autres séquences de messages. Une seule séquence peut se produire à la fois.
- **loop** : Le fragment est répété un certain nombre de fois. Dans la protection, on indique la condition sous laquelle il doit être répété.
- **break** : Si ce fragment est exécuté, le reste de la séquence est abandonné. Vous pouvez utiliser la protection pour indiquer la condition dans laquelle la rupture se produira.
- **par** (parallel) : Les événements des fragments peuvent être entrelacés.
- **critical** : Utilisé dans un fragment par ou seq. Indique que les messages de fragment ne doivent pas être entrelacés avec d'autres messages.
- **seq** : Il existe au moins deux fragments d'opérande. Les messages impliquant la même ligne de vie doivent se produire dans l'ordre des fragments. Lorsqu'ils n'impliquent pas les mêmes lignes de vie, les messages des différents fragments peuvent être entrelacés en parallèle.
- **strict** : Il existe au moins deux fragments d'opérande. Les fragments doivent se produire dans l'ordre donné.

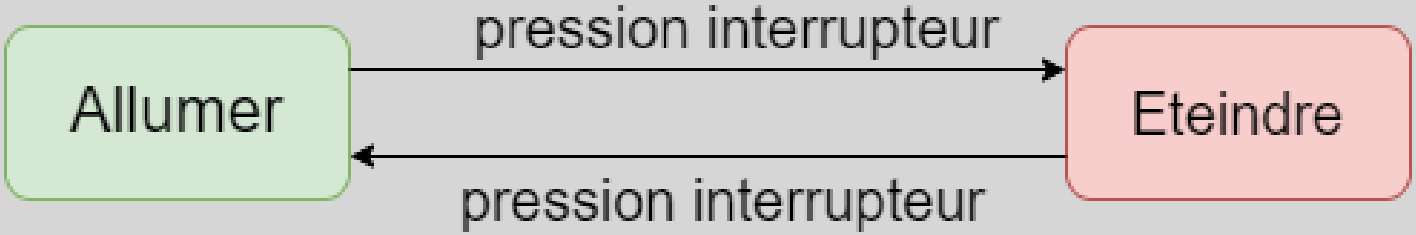
Opérateurs d'interprétation de la séquence

consider : Spécifie une liste des messages que ce fragment décrit. D'autres messages peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.
ignore : Liste des messages que ce fragment ne décrit pas. Ils peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.
assert : Le fragment d'opérande spécifie les seules séquences valides. Généralement utilisé dans un fragment Consider ou Ignore.
neg : La séquence affichée dans ce fragment ne doit pas se produire. Généralement utilisé dans un fragment Consider ou Ignore.

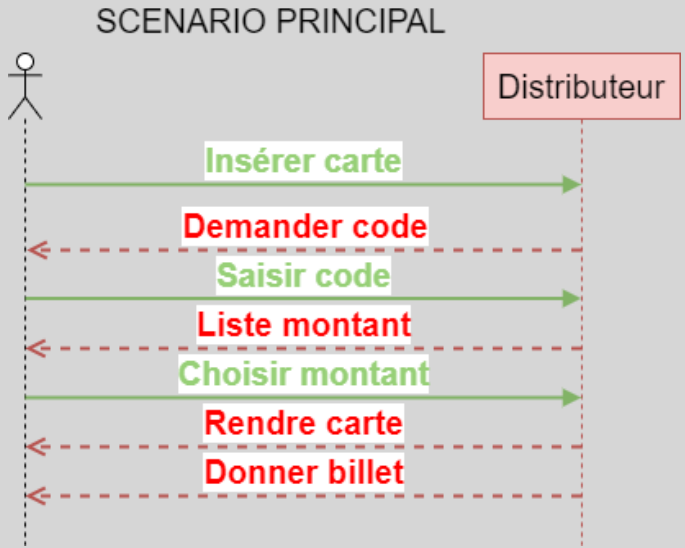


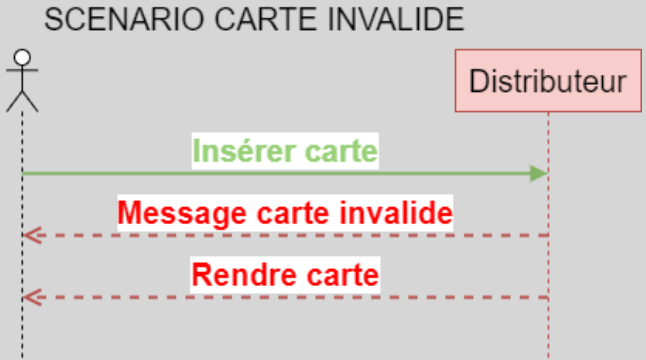
1.9 Diagramme d’état transition

L’objectif est de décrire le comportement dynamique d’une entité (système, logiciel,...)
Ce comportement est décrit par des états et des transitions entre ces états.
cela donne une vue synthétique de la dynamique de l’entité et cela permet de regrouper un ensemble de scénario.



Exemple de plusieurs scénario d'un distributeur





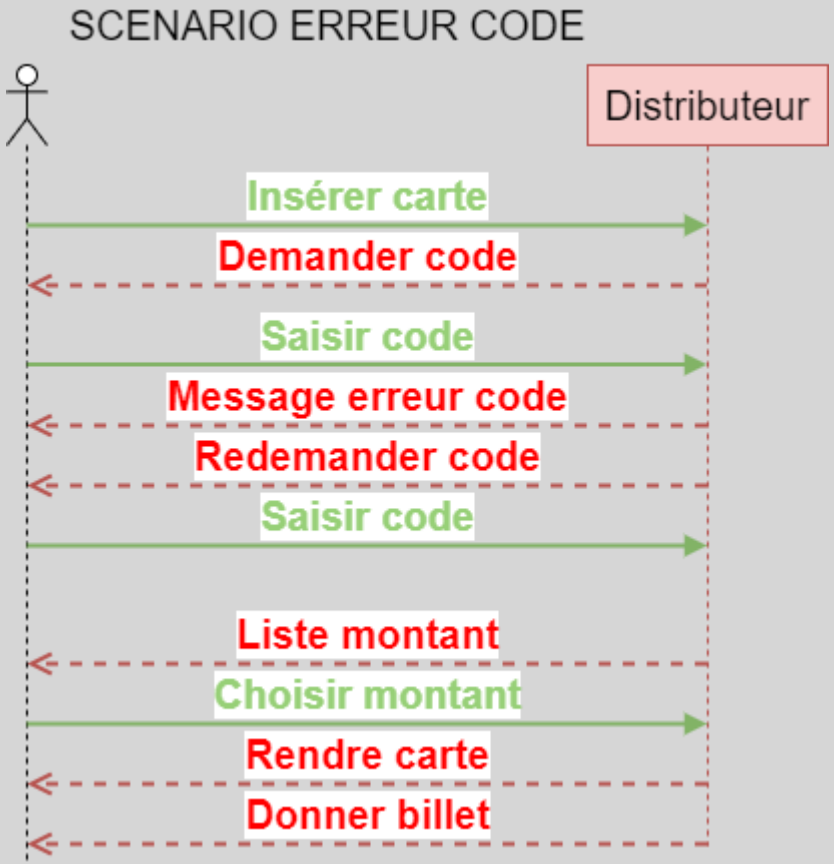
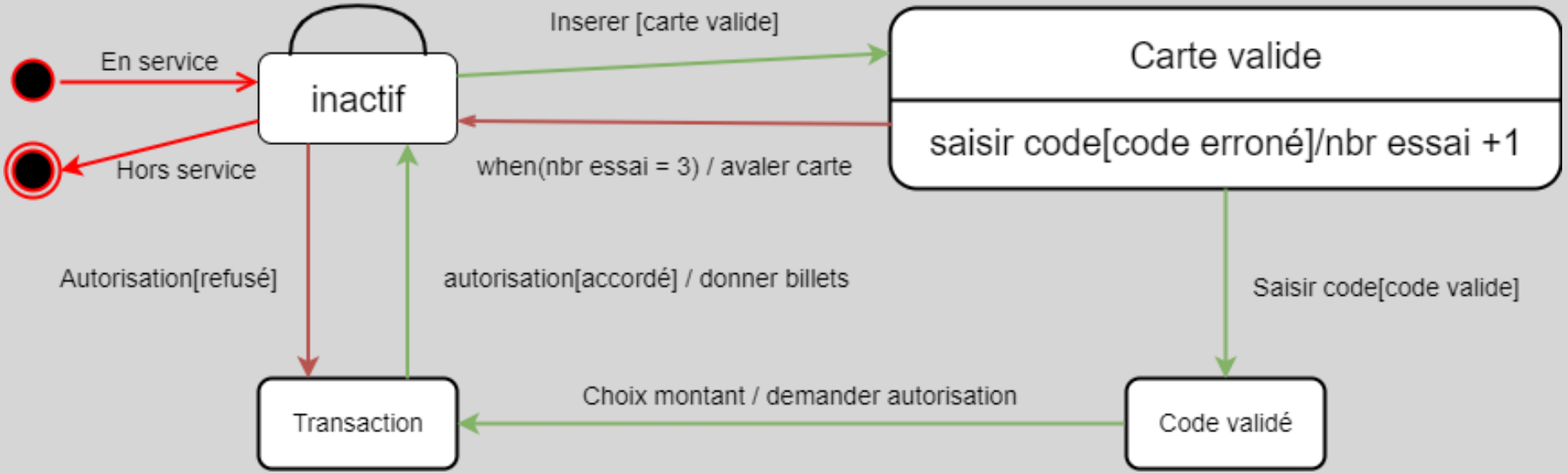
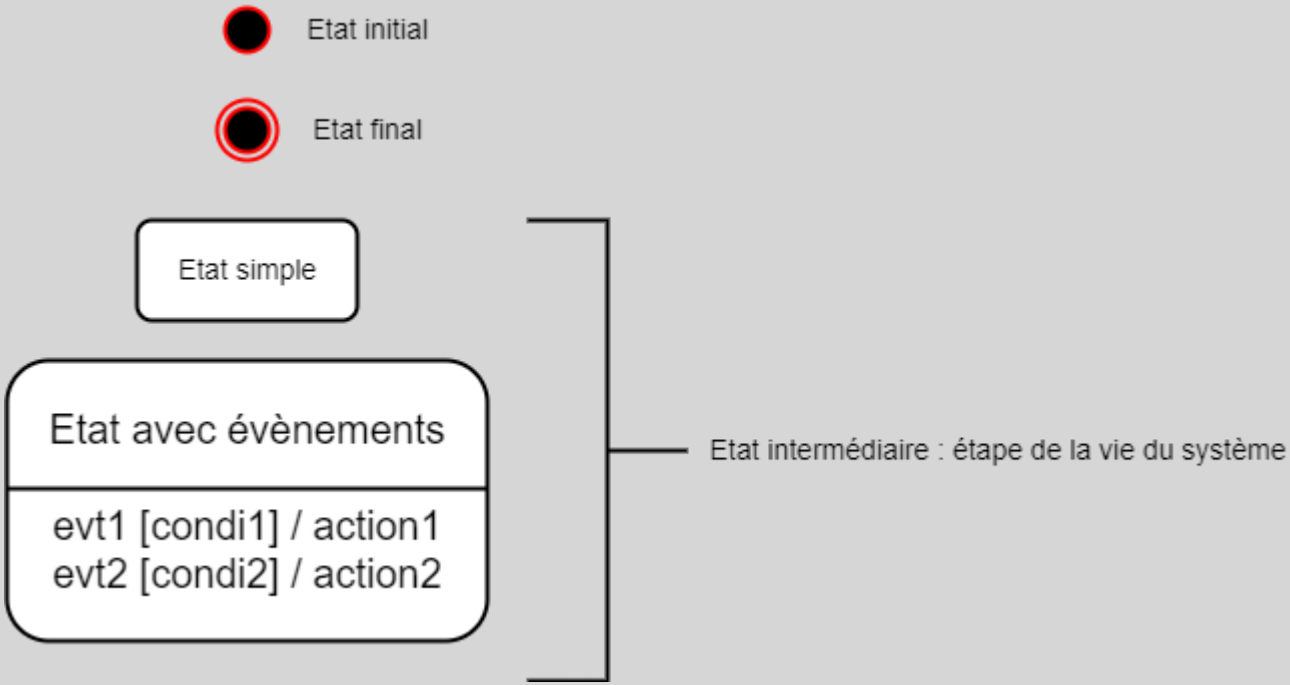


Diagramme d’état transition



Légende des états:



Lexique:

Caractéristique d'un état :

- condition vérifié
- événement attendu (interne ou externe)

Événement :

Fait instantané venant de l'extérieur du système et survenant à un instant donné

Types d'événements :

- Signal : réception d'un message asynchrone
- Appel d'une opération (synchrone) : liée aux cas d'utilisation, opération du diagramme de classes...
- Satisfaction d'une condition booléenne : when(cond), évaluée continuellement jusqu'à ce qu'elle soit vraie
- Temps:
  - Date relative : when(date = date)
  - Date absolue : after(durée)

Action : → Événement [condition] / action

Réaction du système à un événement

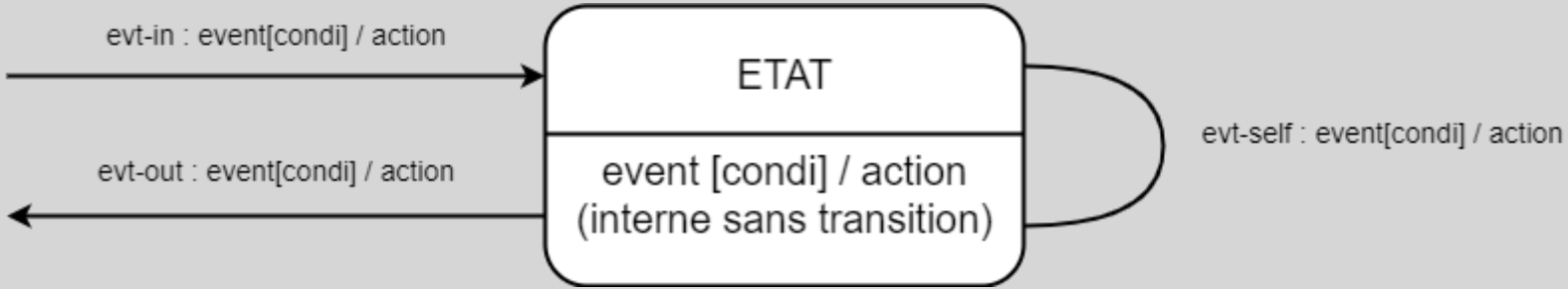
Exemples d'actions (syntaxe laissée libre) :

- affectation
- envoi d'un signal
- appel d'une opération
- création ou destruction d'un objet

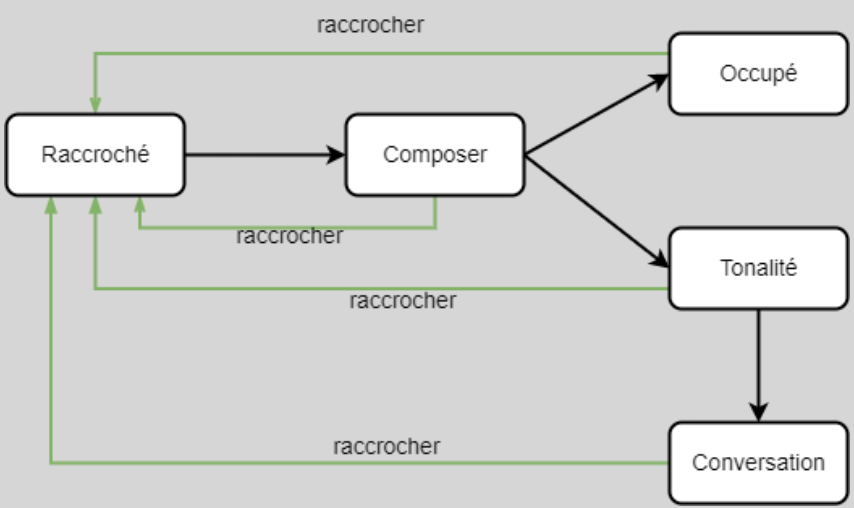
Lexique (suite):

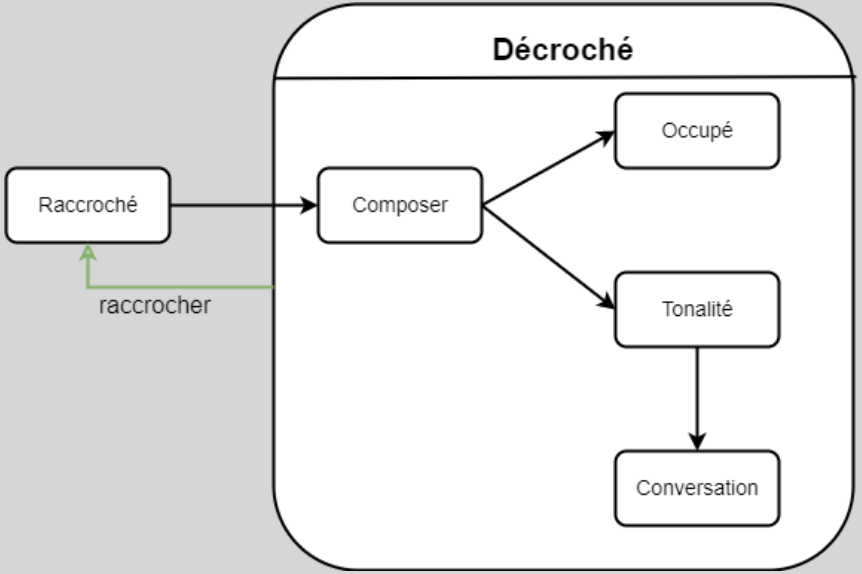
Événements externes à l'état : transitions

- Transition vers l'état : evt-in
- Transition depuis l'état : evt-out
- Transition depuis l'état vers lui-même : evt-self



Etat composite : Etat regroupant un ensemble d'états





1.10 Les autres diagrammes

Les diagrammes de composants :
Ce diagramme permet de mettre en évidence les dépendances entre les composants

Les diagrammes de déploiements :
Ils permettent de montrer la disposition physique des matériels composant le système