Ingénierie du Logiciel
Master 1 Informatique – 4I502

Cours 2 : Analyse
Cas d'utilisation, Classes métier

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Rappel Cours 1

Une méthode définit :

- Quelles informations capturer/spécifier/coder
- Comment réaliser la capture, directives précises de modélisation
- Rôles : qui doit réaliser quoi, définition des activités du développement
- Donner des moyens de gérer la complexité des projets
- Des outils et directives liés à la conduite du projet

Objectifs :

- Améliorer la productivité
- Améliorer la qualité(s) du logiciel

I. Introduction au cycle en V

Le cycle en V

Choix d'une méthode à appliquer dans l'UE : le cycle en V.

- Intérêt pédagogique :
  - très bien spécifié (pas toujours le cas),
  - très classique (positionnement),
  - bonne séparation des tâches et des intentions de chaque tâche (axes du raisonnement)
- Défauts :
  - peu itératif ou itérations longues,
  - plutôt adapté à des projets de grande taille (mise en place assez lourde)

Présentation de ce cycle sur 6 séances.

Utilisation de la notation UML

Le V

Client

Définition du Besoin

Cahier des Charges

Branche Elaboration

Branche Validation

Maintenance

Analyse

Tests de Validation

Rupture du cycle de vie

Conception Architecturale

Tests d'Intégration

Conception Détaillée

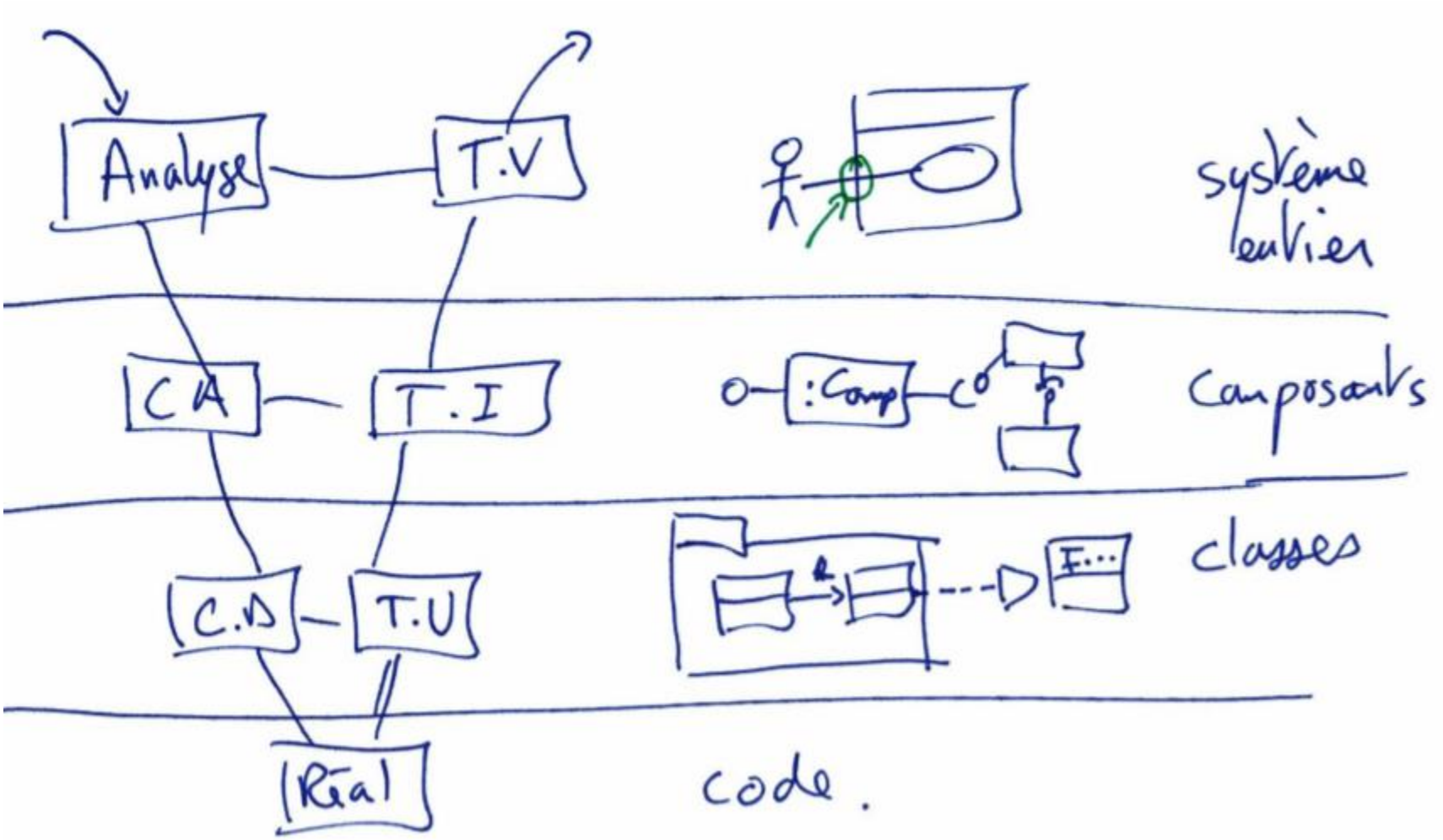
Tests Unitaire

Réalisation

Le cycle en V

- Définition du besoin et phase de maintenance hors de notre étude
  - Le cahier des charges est donc complet
  - Nous sommes une équipe de développement logiciel, constituée d'experts en informatique mais pas experts dans le *métier* du client
  - On s'arrête à la première *recette* (livraison du logiciel fonctionnel)
- Deux branches : élaboration vs contrôle
  - Chaque étape du développement a son pendant permettant de
    - définir les critères de sa réussite
    - et les valider (tester)
  - ✓ Gros accent sur le contrôle pour assurer la qualité (adéquation fonctionnelle en particulier) du résultat

Le cycle en V


Analyse
T.V
système en vrai
C.A
T.I
:Comp
Composants
C.D
T.U
F...
classes
Real
code.

Le cycle en V

- Niveaux de description/granularité
  - Analyse/TV : système en boîte noire, échanges observables avec l'extérieur uniquement
  - Conception Architecturale/TI : le système est découpé en composants (opaques), on étudie les échanges entre ces composants pour répondre au besoin utilisateur.
  - Conception Détaillée/TU : on étudie la réalisation de chaque composant, avec une description OO au niveau classes.
  - Réalisation : niveau code (peu abordé dans l'UE).

La phase d’analyse

L’analyse

Trois éléments clés dans l’analyse d’un système (Sommerville):

- Requirements Definition – Cahier des Charges
  - Etabli par le client
  - En langage naturel (informel, lacunaire, jargon métier…)
  - Définit les services attendus (offerts) du système et les contraintes de fonctionnement
- Requirements Specification – Spécification Technique du Besoin
  - Spécification, donc structuré, détaillé, précis, complet
  - Définition *fonctionnelle* du besoin
  - Notion contractuelle du document
- Software Specification – Description d’architecture logicielle
  - Description de la *solution* logicielle
  - Reste abstrait mais est une base importante pour la conception et réalisation
  - Détaille et raffine la STB e.g. en explicitant les traitements ou en prévoyant leur orchestration

Analyse vs Conception

Globalement : Séparer le « quoi » du « comment »

- E.g. Le téléphone, quoi évolue peu, comment évolue très vite.
- Le « quoi » est plus proche du *métier*, donc soumis à des contraintes (et habitudes) des utilisateurs
- Le « comment » est plus dépendant de *notre* métier, donc des technologies et solutions sur lesquelles on s'appuie pour *réaliser* le quoi

Spécifier le « quoi » avant de commencer à concevoir et réaliser c'est le but de l'analyse.

- En ingénierie classique (e.g. BTP) c'est même complétement obligatoire de procéder ainsi. En IL c'est plus discutable, cf *agilité*.

L’étape analyse dans le cycle en V de l’UE

- On procède à une instanciation particulière d’une méthode de développement basée sur le V et sur UML dans l’UE.

L’analyse découpée selon deux axes : Fonctionnel et Structurel

- Axe fonctionnel :
  - Services observables rendus par le système à son environnement
  - Signatures de ces échanges
- Utilisation de :
  - Diagrammes de cas d’utilisation
  - Diagrammes de séquence
  - Fiches Détaillées de cas d’utilisation (langage naturel structuré)
  - Tests de validation (langage naturel structuré)

Axe structurel

Axe structurel :

- Description des données stockées par le système et qui constituent son état : diagramme de classes *métier*
  - ✓ /!\ pas des classes au sens OO ou pour implanter
  - ✓ /!\ même si ça s'en rapproche plus, pas non plus un schéma de BDD avec des primary key
  - ✓ L'état du système à un instant = décrit par une instance de ce diagramme de classe
  - ✓ Une pré/post condition sur un use case => testable avec les données de ce diagramme

Analyse

Donc 5 « délivrables » pour un cahier des charges

- Diagramme de use case
- Fiches détaillées de ces use case

- Séquences d'interaction
- Tests de validation

- Diagramme de classes *métier*

Constitue le sujet de l'examen réparti 1 (Novembre) sur un cahier des charges nouveau chaque année. Cf Annales.

II. UML
Cas d’utilisation, Classes

UML

- Une **notation** unifiée et standard pour décrire les artefacts logiciels
  - Pas de process logiciel, pas de méthode associée
- Beaucoup de diagrammes, avec chacun divers niveau d'abstraction qu'on peut adopter (informel -> génération de code)
- Dans cette UE :
  - Diagrammes de cas d'utilisation,
  - Diagrammes de classe et composant,
  - Diagrammes d'objet et de structure interne
  - Diagrammes de séquence
- Attention la méthode => une certaine utilisation d'un sous ensemble des concepts offerts par UML dans un but précis

Diagramme de Cas d'Utilisation (Use case)

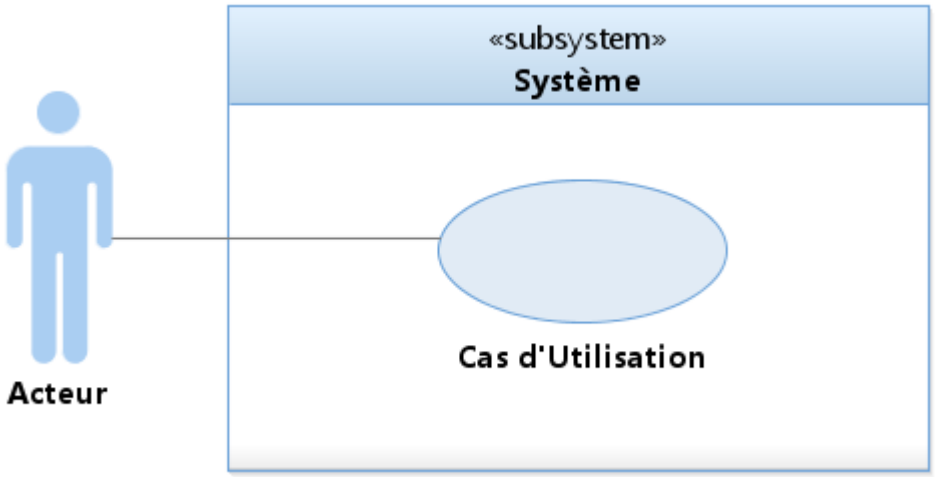
But/Axe :

- Identifier les *missions* du système
- Identifier les frontières du système
- Identifier les utilisateurs et leur rôle vis-à-vis du système
- Permettre la discussion avec le client, introduire une découpe *fonctionnelle* du système

Diagramme de Cas d'Utilisation (Use case)

Des diagrammes relativement simples

- Trois entités :Système, Acteur, Cas d'Utilisation
- Un seul lien simple entre acteur et cas d'utilisation : communique
- Deux liens possibles entre cas d'utilisation (include, extend), héritage possible entre acteurs

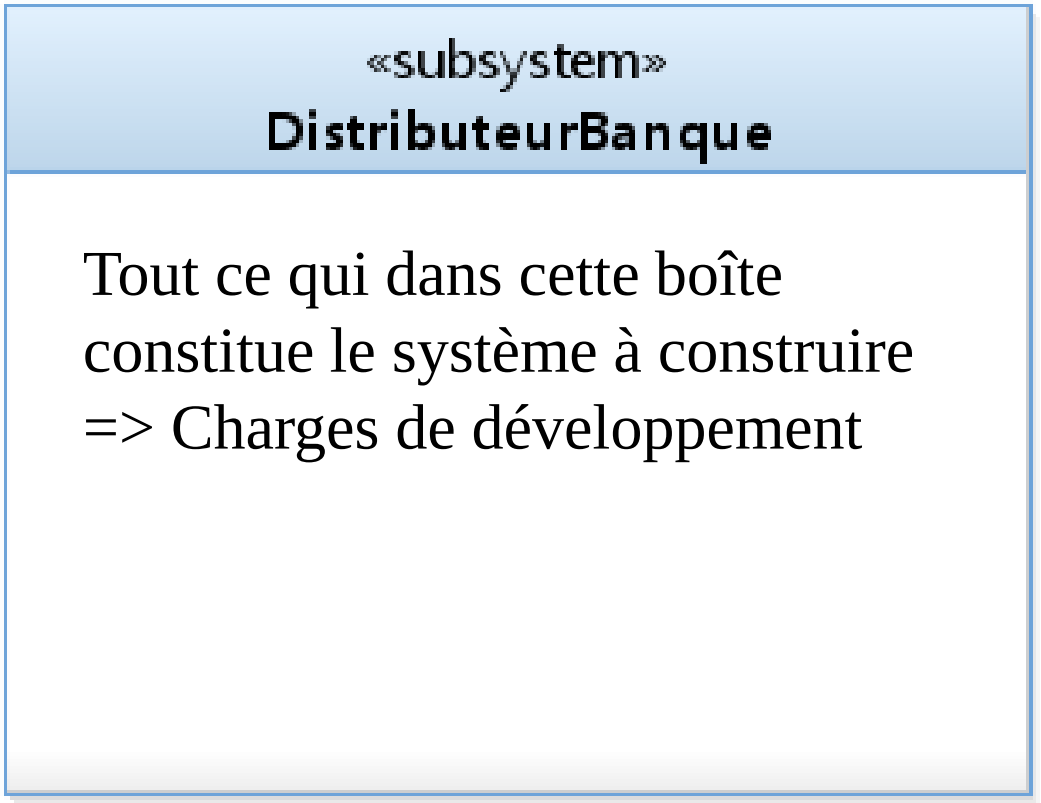


Système

Pose le contexte et les frontières du système.

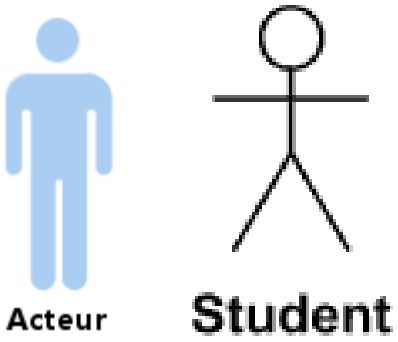
Techniquement stéréotype <<subsystem>>

Tout ce qui à l'extérieur => existe déjà, correspond à des éléments que l'on ne va pas développer.



Acteur

Définition Entité **externe** au système, qui **interagit** avec lui. L'acteur joue un rôle particulier vis-à-vis du système. Une même entité physique peut être représentée par plusieurs acteurs et réciproquement.



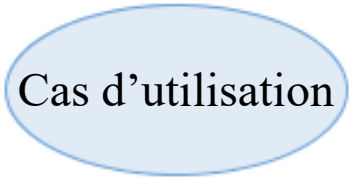
Attention l'acteur *doit* interagir avec le système !

- Caisse enregistreuse classique : le client du magasin est-il un acteur ?
- Caisse automatique : le client est-il un acteur ?
- Le caissier occupe-t-il le même rôle dans une caisse classique et une caisse automatique ?

Cas d'utilisation

Définition Un cas d'utilisation ou use case représente **une action de bout en bout** qui **apporte quelque chose** aux acteurs. Un cas d'utilisation se décrit comme **un ensemble d'interactions** entre un acteur et le système, déclenché par l'acteur. Un cas d'utilisation peut préciser des **pré et/ou post conditions** qui portent sur l'état du système avant et après un déroulement réussi du cas d'utilisation.

Cas d'Utilisation



Nommage du cas d'utilisation : utiliser un verbe, formuler du point de vue de l'acteur.

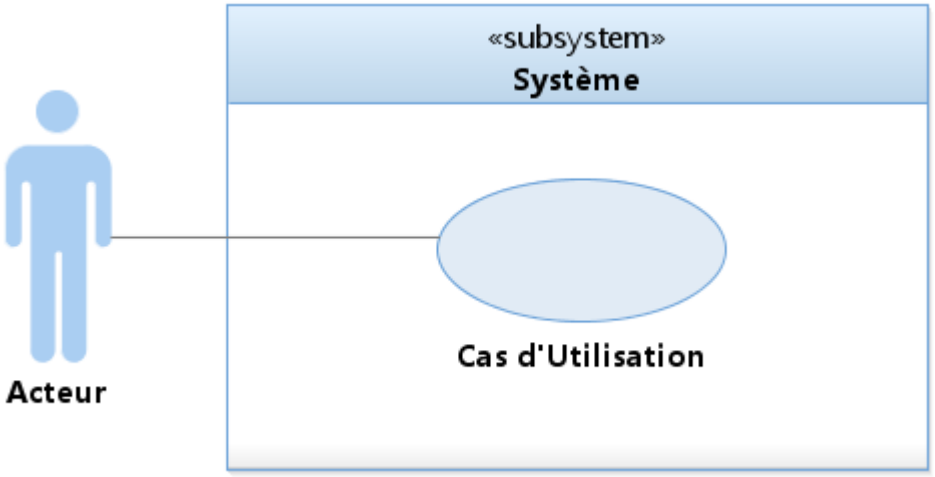
- Distribuer des billets -> Retirer de l'argent

Attention à la granularité : *ensemble de scenarios* d'interaction…*de bout en bout*

- Approuver EULA, Jouer un coup… pas des « cas d'utilisation » (sauf si échecs en différé ?)
- L'utilisateur se connecte au système *dans l'objectif de* faire le cas, puis pourrait se déconnecter, son objectif est atteint.

Relation Communique

Relie un acteur à un cas d'utilisation en franchissant la frontière du système.

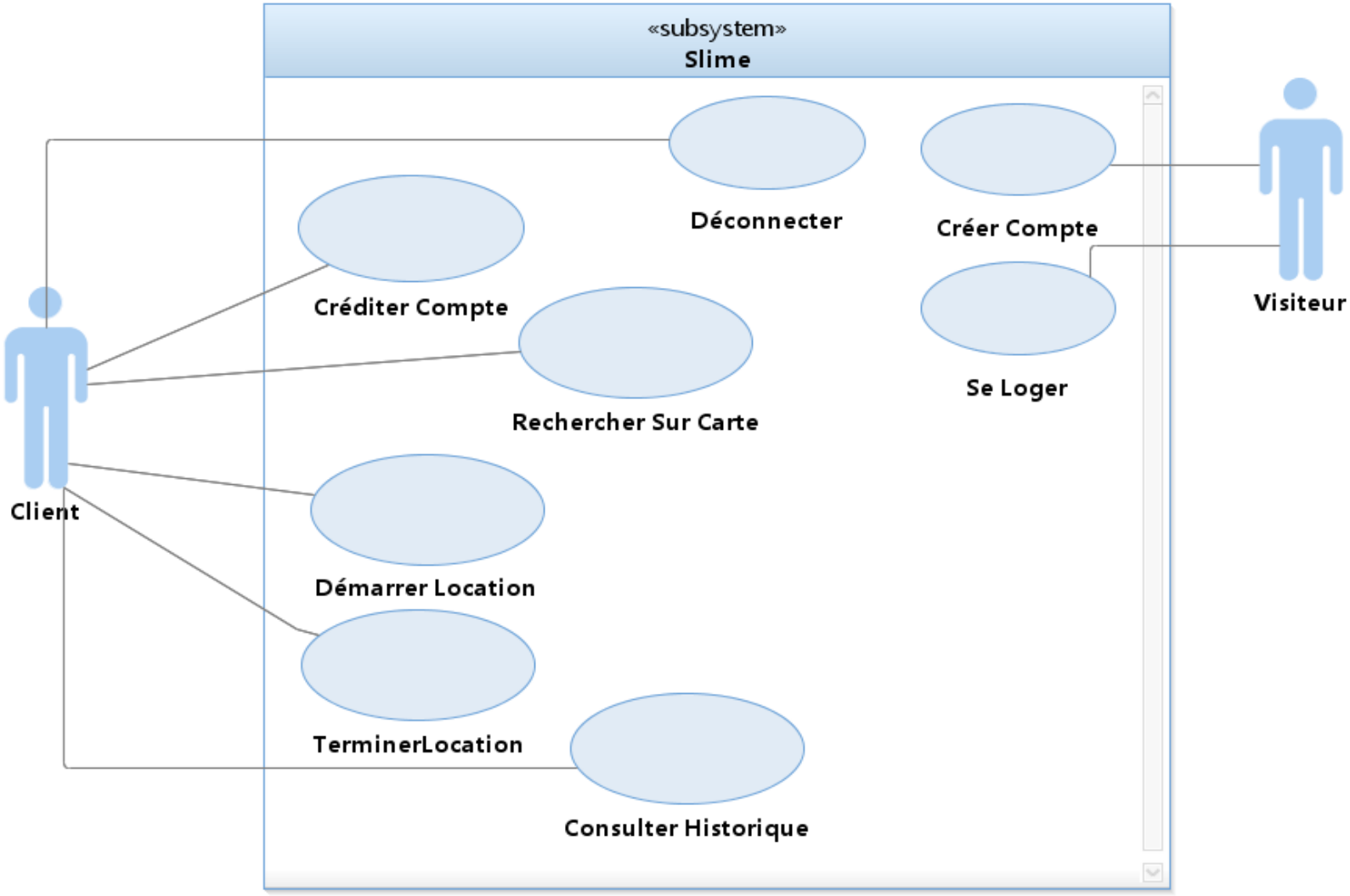


Représente une communication entre l'acteur et le système dans la réalisation de ce cas d'utilisation.

Simple trait plein, sans flèche d'orientation.

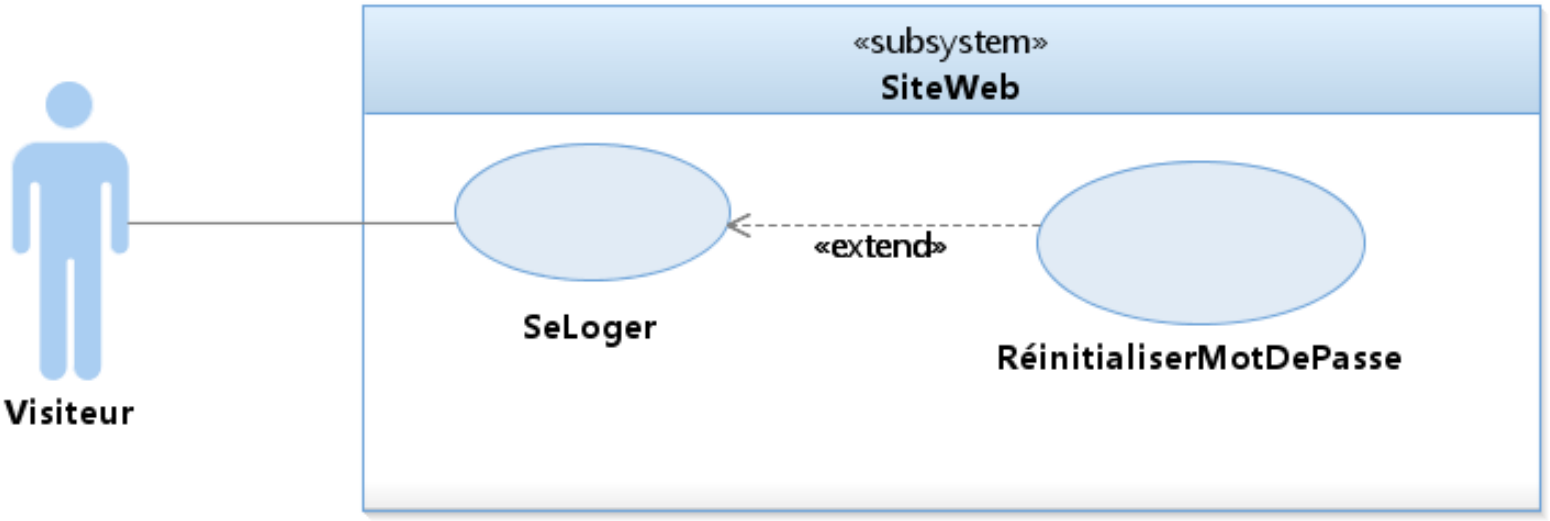
UML permet de poser des multiplicités aux extrémités mais on évitera dans l'UE.

Exemple


«subsystem»
Slime
Déconnecter
Créer Compte
Visiteur
Créditer Compte
Se Loger
Rechercher Sur Carte
Client
Démarrer Location
TerminerLocation
Consulter Historique

Use Case avancés : <<extend>>

A -- <<extend>> -> B : Au cours du déroulement de **B**, **parfois (de façon optionnelle),** une étape est de réaliser les interactions décrites dans le use case **A**.

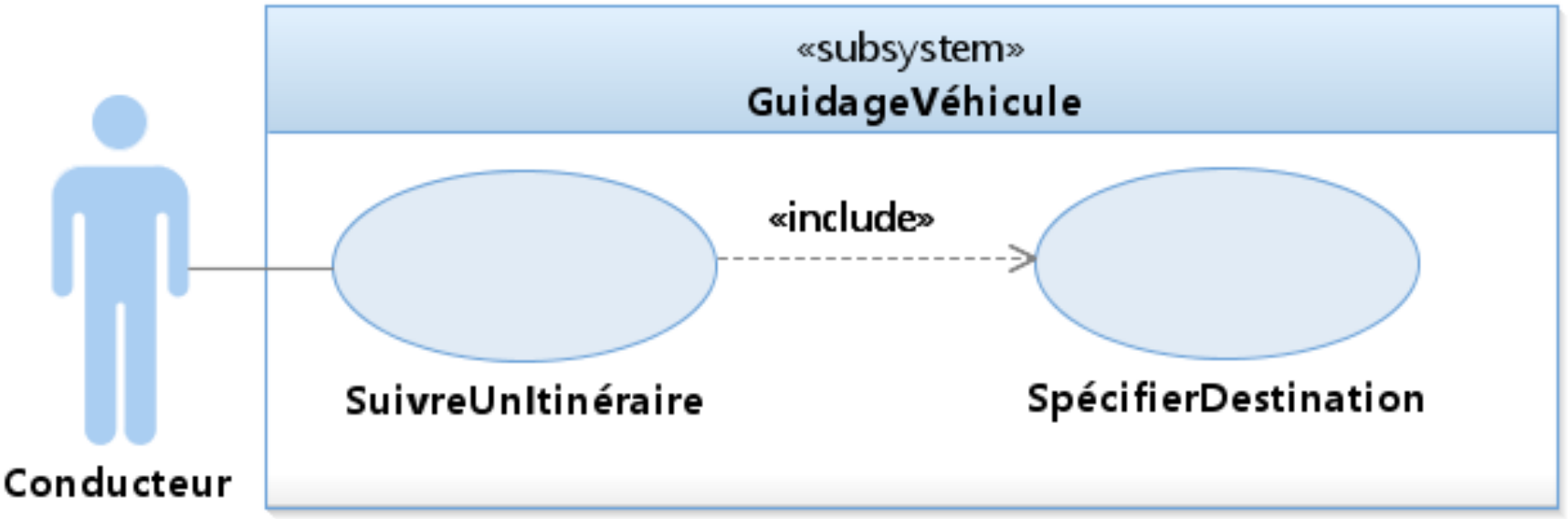


"Parfois, pendant que j'essaie de me loger, il faut réaliser les interactions me permettant de réinitialiser mon mot de passe"

- Attention à la granularité A et B sont des *cas d'utilisation* à part entière !
- /!\ Direction : A étend B : c'est B le cas d'utilisation principal
- Un *scenario alternatif (ALT)* de B consiste en « réaliser A »
- UML strict => notion de point d'extension dans B, pas utilisé dans l'UE.

Use Case avancés : <<include>>

A -- <<include>> -> B : Au cours du déroulement de **A**, une étape **obligatoire** est de réaliser les interactions décrites dans le use case **B**.



"A **chaque fois** que je veux aller quelque part, une étape consiste à spécifier ma destination.

- Attention à la granularité A et B sont des *cas d'utilisation* à part entière !
- /!\ Direction : A inclut B : c'est A le cas d'utilisation principal
- Le scenario *nominal (SN)* de A contient une étape « réaliser B »

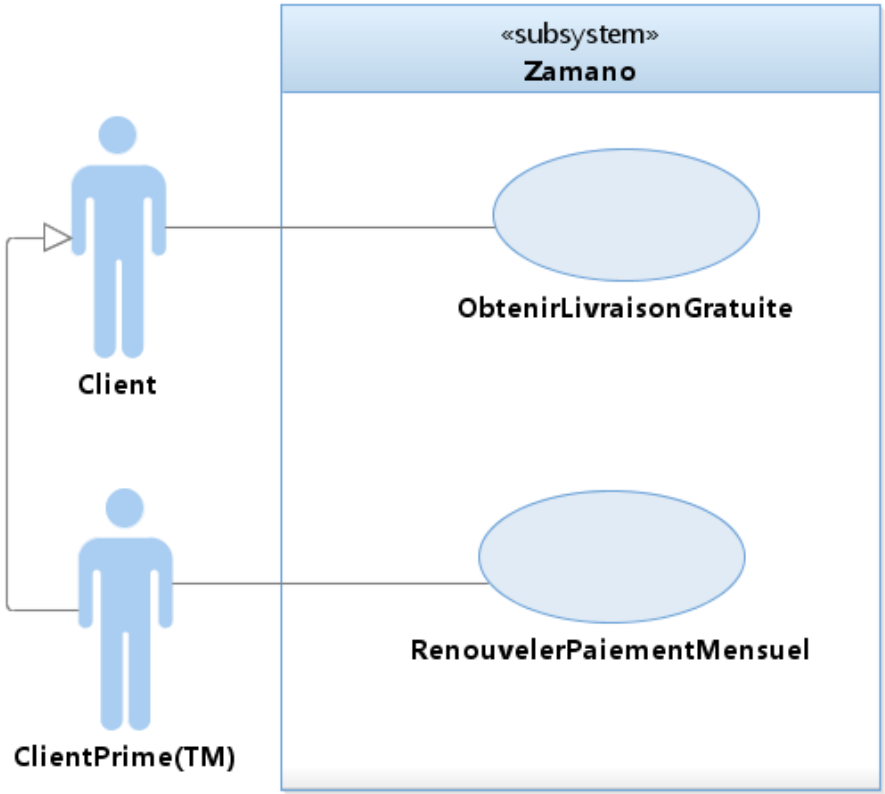
(Use case et authentification)

- Version include
  - Tous les UC nécessitant authentification <<include>> s'authentifier
  - Notion de session ? Surcharge du diagramme ?
- Version rôle (diagramme précédent)
  - En fonction du nom d'acteur, différentes fonctionnalités offertes
  - Une action s'authentifier offerte à tous, on ajoute se déconnecter par homogénéité
  - La spécification précise se fait dans les pré-conditions des cas d'utilisation, e.g. UC Modifier note, « Pré-condition : L'utilisateur est authentifié en tant qu'enseignant »

Héritage entre acteurs

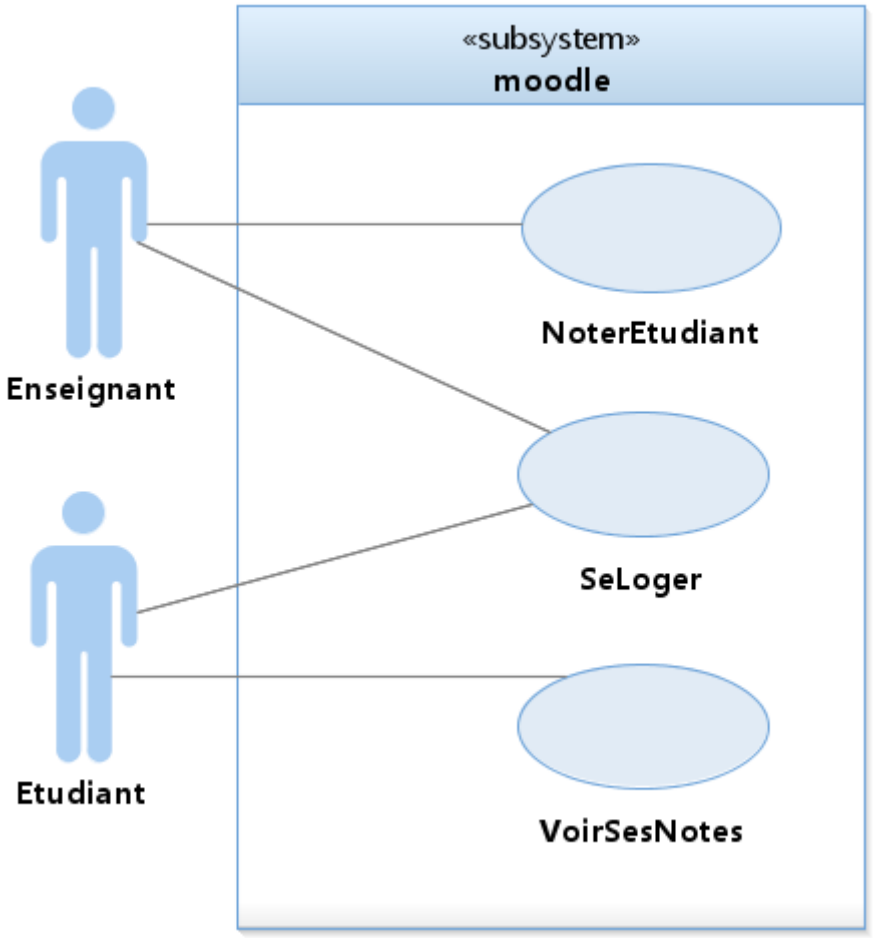
Notion de hiérarchie de droits ou permissions sur le système

- Le superviseur peut faire tout ce que fait le caissier, mais il peut aussi annuler des produits sur les factures.
- Lien de *généralisation* UML : grosse tête triangle fermée, se lit « est un »
- Attention au sens métier : éviter de noter que ClientPrime “est un” client basique. Ces rôles servent à l’interface avec le client.

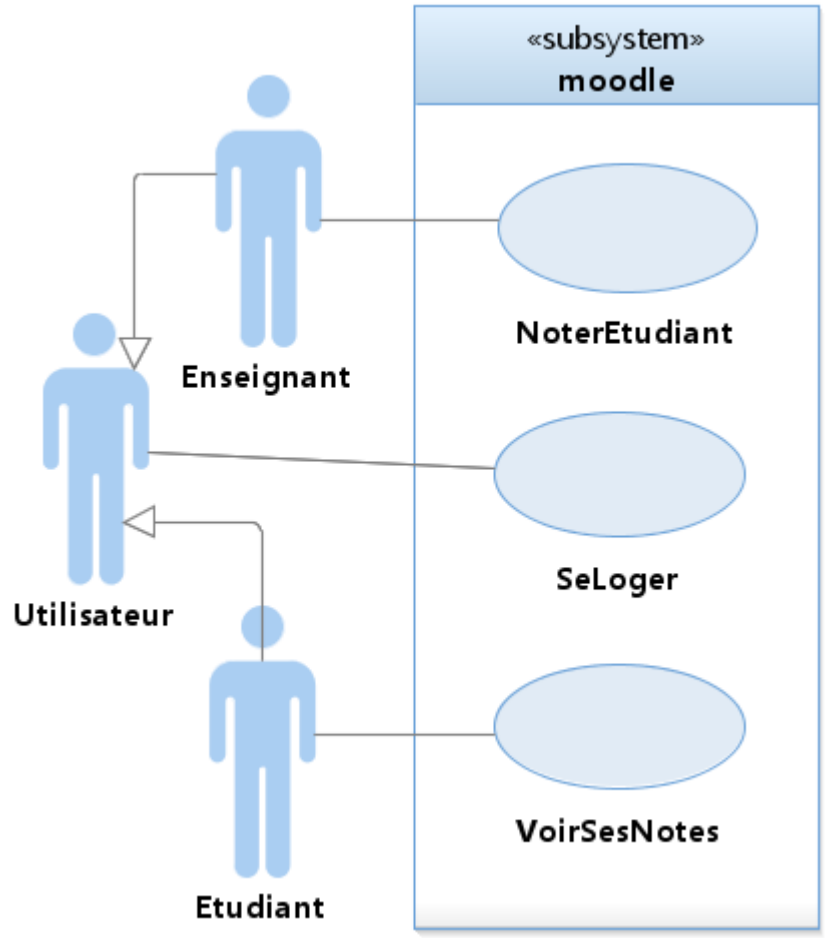


Héritage vs. Double lien

Une même action est disponible pour deux acteurs



(bof, ambigü)



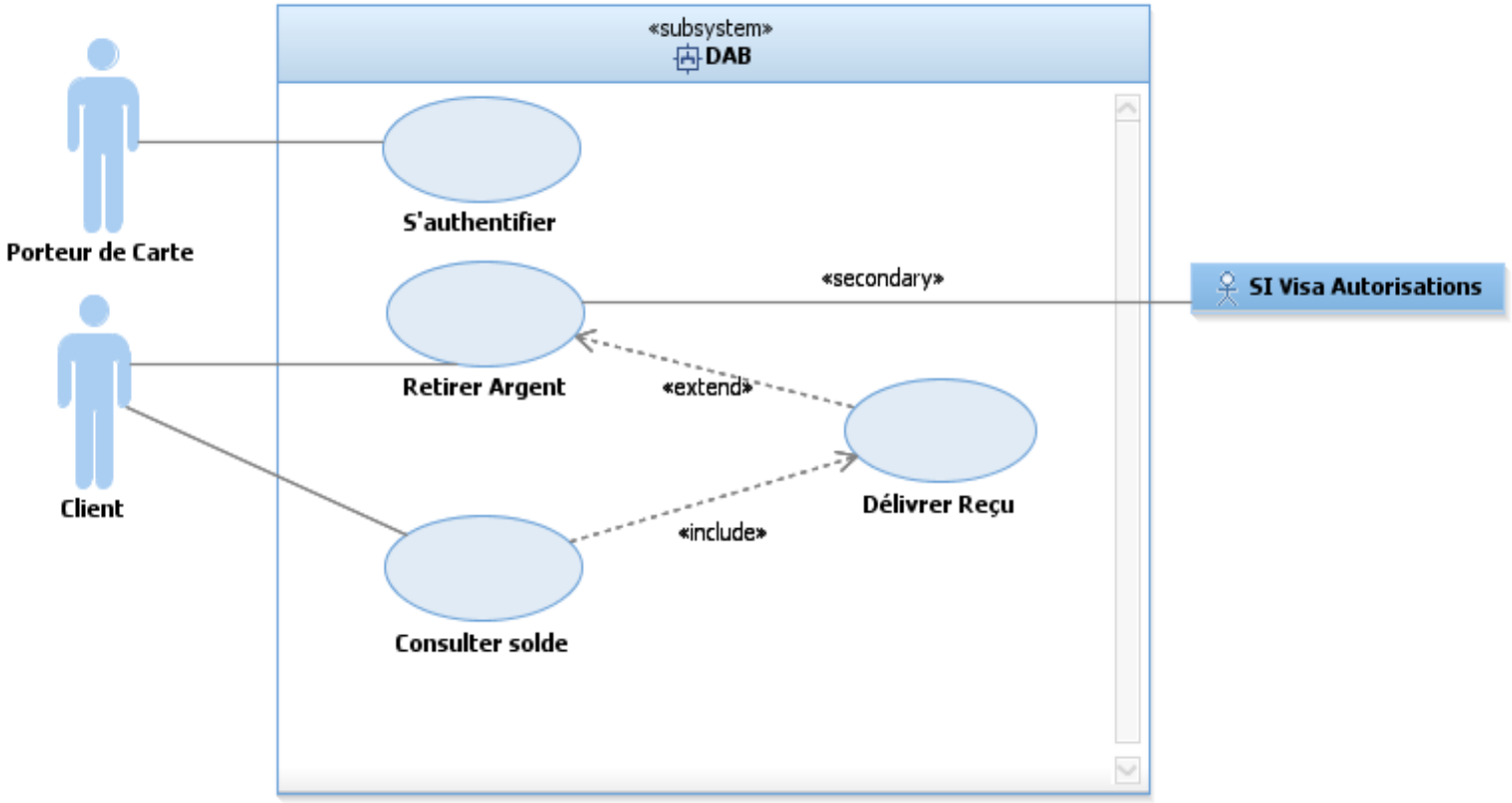
préférable

Use case avancé : acteur et relation secondaire

Certains acteurs = autres systèmes existants plutôt que des humains

Les entités que l'on ne développe pas mais avec lesquelles on interagit méritent leur modélisation.

Si c'est le système qui sollicite l'acteur, on parle d'interaction secondaire, ou on dit que l'acteur est secondaire (dans l'interaction)



Use case avancé : traitements périodiques

Certains traitements sont déclenchés automatiquement

- À une certaine date, tous les soirs à minuit, quand une autre condition externe se produit…
- Le système accomplit une mission/un traitement à cet instant mais pour le compte de quel acteur ? Un cas d'utilisation sans acteur, ça ne peut pas exister, par définition.
- Solution : introduire un acteur artificiel « chron », « autosave », « tache quotidienne » …
- C'est cet acteur qui fait le *trigger* ou première étape qui déclenche le use case.

Use Case : le reste

On pourra croiser dans d'autres supports :

- Héritage entre cas d'utilisation
  - Exemple : payer -> payer CB, payer liquide, payer téléphone
  - Payer basket « est une » façon de payer ou pas ?
  - On évite ces liens d'héritage entre use case dans l'UE.
- Des multiplicités sur les associations acteur/use case
  - Dans un but de simplicité, on évite dans l'UE
- Des flèches (vers le use case) sur les liens de communication : incorrection basée sur les versions 1.x du standard UML, à éviter.

Use case : granularité/abstraction

- L'objet « subsystem » donne le *scope* de l'étude
  - Si c'est le système entier, use case = scenario de bout en bout des utilisateurs finaux (acteurs)
  - Si c'est un composant, use case = scenario d'utilisation de l'API du composant dans une interaction avec d'autres composants logiciels (acteurs)
  - Si c'est une classe, use case = scenario ou plus simplement méthode offerte, l'acteur est un client de la classe i.e. tout code qui l'utilise.

UML Classes Métier

Le diagramme de classe métier

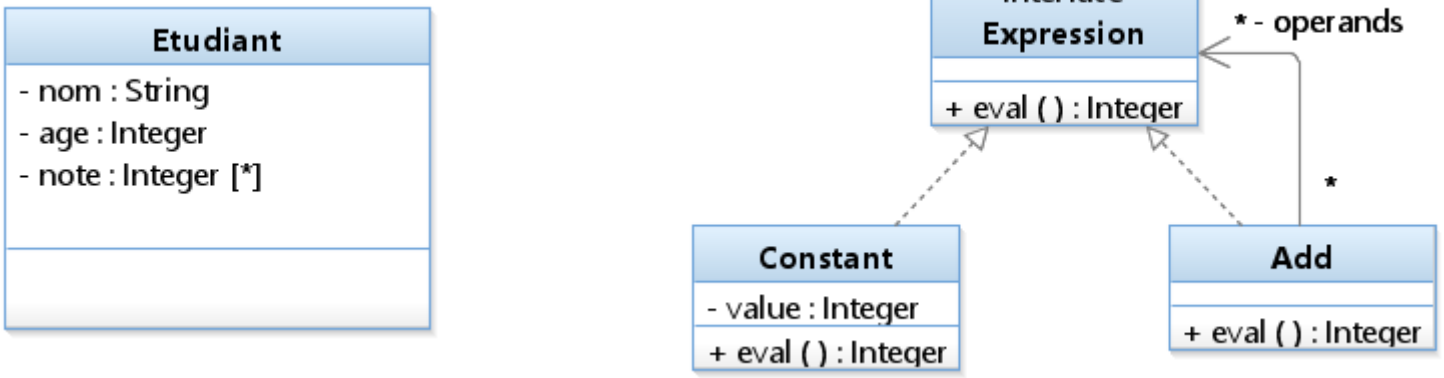
Granularité analyse : le système est une boite noire

- Toutes les données sont implicitement visibles du système ; le système lui-même n'est pas représenté (il englobe)
- Pas d'acteurs sur le diagramme (!), pas de dynamique (session)
- Pas d'attribution de responsabilités aux données (méthodes) ; ce ne sont pas des classes au sens Java
- Héritage limité pour les nécessités du typage en commun (e.g. données arborescente)
- Pas d'aggrégations, compositions, navigabilités et autres précisions relevant d'une réalisation particulière.

- Typage des attributs uniquement types simple : numériques, bool, string, date et [*] de ces types
- Multiplicités importantes, noms de rôles sur les associations

Diagramme de classe UML

Diagramme de classe



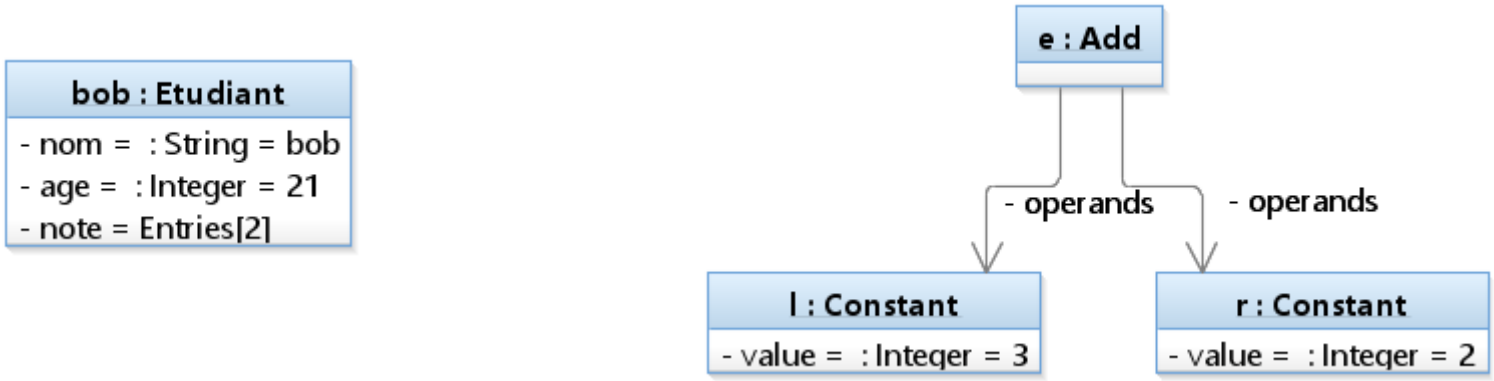


Diagramme des objets

Classe et instance

- Classe représente un type d'entité
  - Associations = liens potentiels entre instances de classes
  - La classe peut s'instancier un nombre arbitraire de fois
- Diagramme des objets
  - Snapshot / photo instantané de l'état de l'application à un instant donné (mémoire heap d'un programme Java ?)
  - Les instances de classes sont liées d'une manière spécifique, mais qui obéit aux règles de « grammaire » du diagramme de classe
- Diagramme de classe métier =>
  - Pas de modélisation des méthodes, description simplifiée des associations
  - Un diagramme des objets instance de ce diagramme = un *état* de l'application, i.e. qui peut influencer perceptiblement les échanges futurs avec l'extérieur

La classe : compartiments

Compartiments d'une classe :

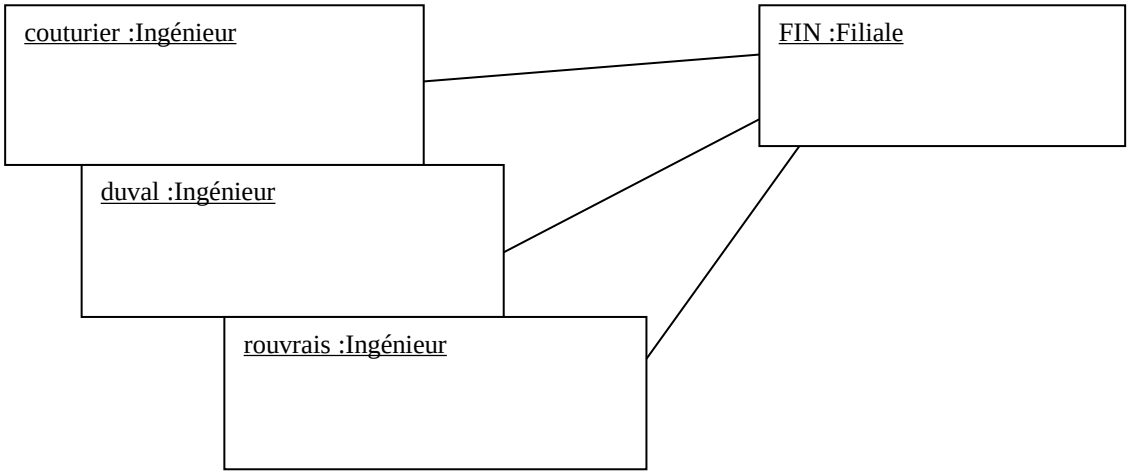
- Nom et stéréotype (<<interface>>,<<abstract>>…)
- Attributs
  - ***vis nom : type***
  - Visibilité : +public, -private, #protected, ~package
  - Type = String, Integer, Boolean + Date.
    - On préfère une association pour les attributs typés par une autre classe.
  - Multiplicité [*] désigne une liste : e.g. String[*] pour une liste de noms
- Opérations
  - ***vis nom (arg1:type1, arg2:type2) : typeRetour***
  - Idem pour le type multiplicité [*] admise
- Autres compartiments UML optionnels, responsabilités, structure, … pas utilisés dans l'UE.

L'association

Association bi-directionelle sans plus de précisions (pour les CM)

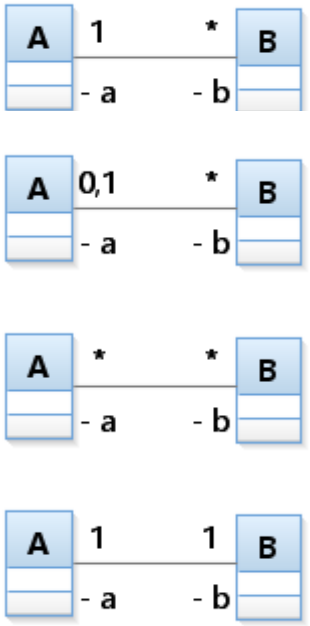
- Trait plein reliant deux classes
- Deux extrémités portant chacune
  - Visibilité, nom de rôle, multiplicité
  - En général en plus : aggregation/composition, navigabilité
- Multiplicités
  - 0,1 : peut être vide (null) ou prendre une valeur
  - 1 : ne peut exister sans ce lien
  - * : nombre arbitraire, e.g. une liste d'éléments
  - On peut affiner plus, e.g. "3,11", mais on ne trouve pas ça particulièrement pertinent dans l'UE

Association multiplicité et instances

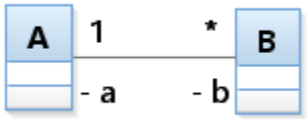


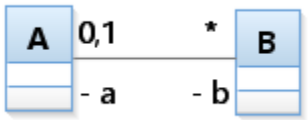
Associations

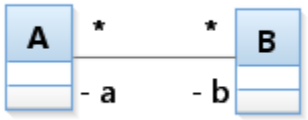
- Exemple A-B -> instances

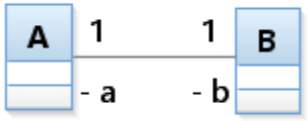


- /!\ liens 1-1 entre deux classes (e.g. Client-Compte) indique fusion possible des concepts


A
1
*
B
- a
- b


A
0,1
*
B
- a
- b


A
*
*
B
- a
- b


A
1
1
B
- a
- b

Association avancées

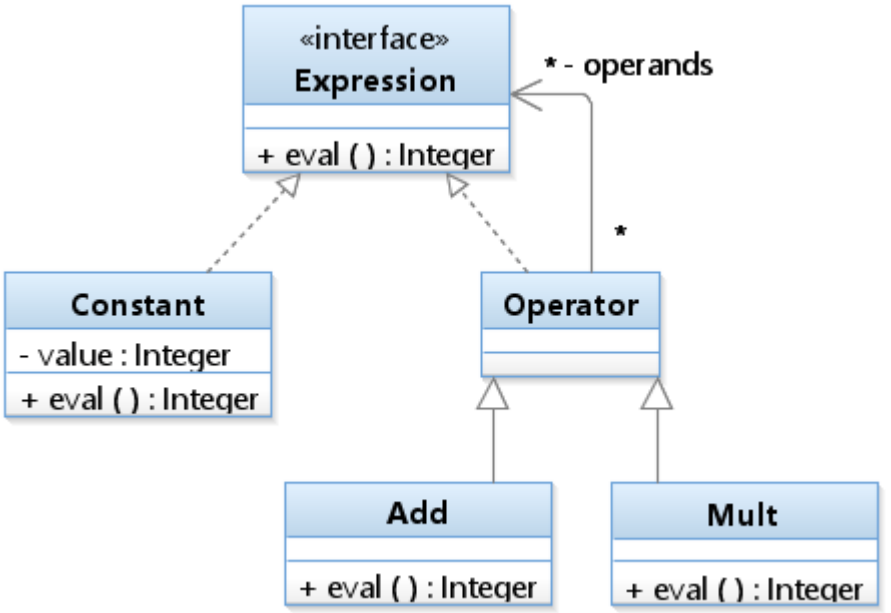
- Association réflexive : source = destination
  - Classe Personne
    - Association conjointDe ?
    - Association parentDe/enfantDe ?
  - Importance des noms de rôle

Association avancées

- Doubles associations
  - Classes UE, Enseignant
  - Associations :
    - EnseigneDans
    - Responsable
  - ✓ De nouveau, importance des noms de rôle et multiplicités.

Généralisation

- Flèche pleine à tête triangulaire fermée : <<generalize>> ou extends en Java
  - Flèche en pointillée avec la même tête : <<realize>> ou implements en Java
- Se lit « est un »
  - Classe parente/classe dérivée, généralisée/spécialisée
  - La classe fille possède toutes les propriétés (attributs, méthodes, associations) du parent en plus des siens
  - L'héritage permet de factoriser
    - des propriétés communes (attributs)
    - des relations en commun (associations)
    - Des déclarations d'opérations (e.g. interface)

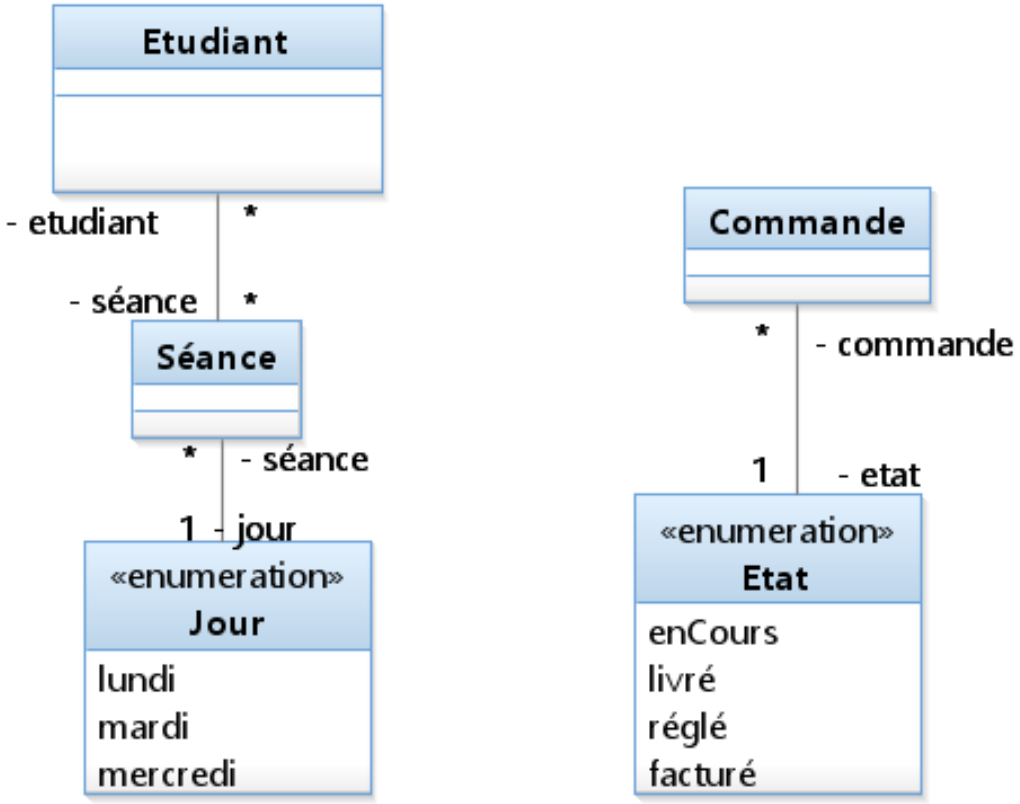


Enumérations

Pratique en modélisation métier

- Ensemble de valeurs possibles dans un domaine fini
  - Techniquement il existe exactement une instance de l'énumération pour chacune de ses valeurs

- Pas des attributs, plutôt des valeurs (litéraux) d 'énumération
- Penser "enum" comme en Java ou C/C++.



Construction des diagrammes

Démarche

- Approche conjointe Structure (classes) et fonction (use case)
  - Fonctions/utilisations du système => use case
  - Structure ou données => classes métier
  - Approche stylo bicolore ou tableau deux colonnes possible

- Le scope = le système, acteurs = utilisateurs extérieurs
- Granularité des use case grossière
  - Ne pas trop tomber dans les details, les fiches détaillées viendront completer la description
  - Etre homogène dans la description
  - Eviter les liens entre use case (les justifier via fiche détaillée)
- En général, 5 à 10 use case, 5 à 10 classes une vingtaine de propriétés.

Identification des use case

- Objectifs :
  - rester gros grain (missions),
  - acteurs vs système (boîte noire),
  - Pas trop d'extend/include (un schéma simple et lisible)
  - ~10 use case maximum dans l'UE
- Approche :
  - Bien lire le cahier des charges, identifier les acteurs cités
  - Souligner tout ce qui ressemble à une action du système
  - Regrouper les actions en les rapprochant des acteurs
  - Réorganiser au besoin (acteurs, découpage des use case) pour éviter les redondances ou les chevauchements de responsabilité

Identification des classes métier

- Objectifs :
  - Capturer les *données métier*
  - Pas d'association dirigée (relève de la conception, e.g. implém BD)
  - Pas d'opérations, donc pas d'interfaces ou autre.
  - Associations bi-directionnelles : bien préciser
    - La multiplicité à chaque extrémité
    - Le rôle à au moins une des extrémités
  - Type et multiplicité des attributs importants :
    - Types simples : Integer, Boolean, Float, Date, String
    - Types complexes => association vers une nouvelle classe
      - On peut utiliser des <<enum>>

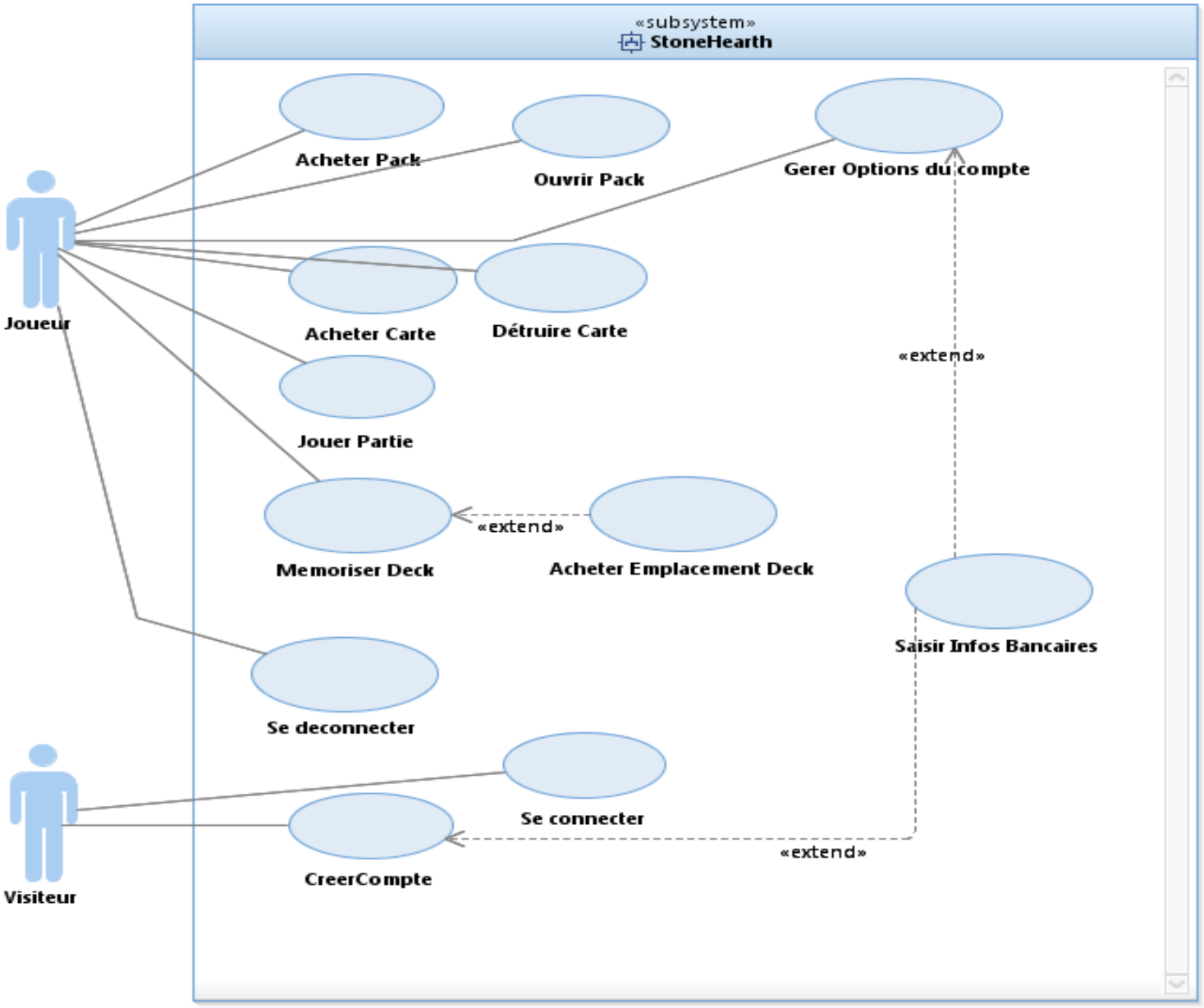
Un bon diagramme de classe métier capture les états possibles du système.

Identification des classes métier

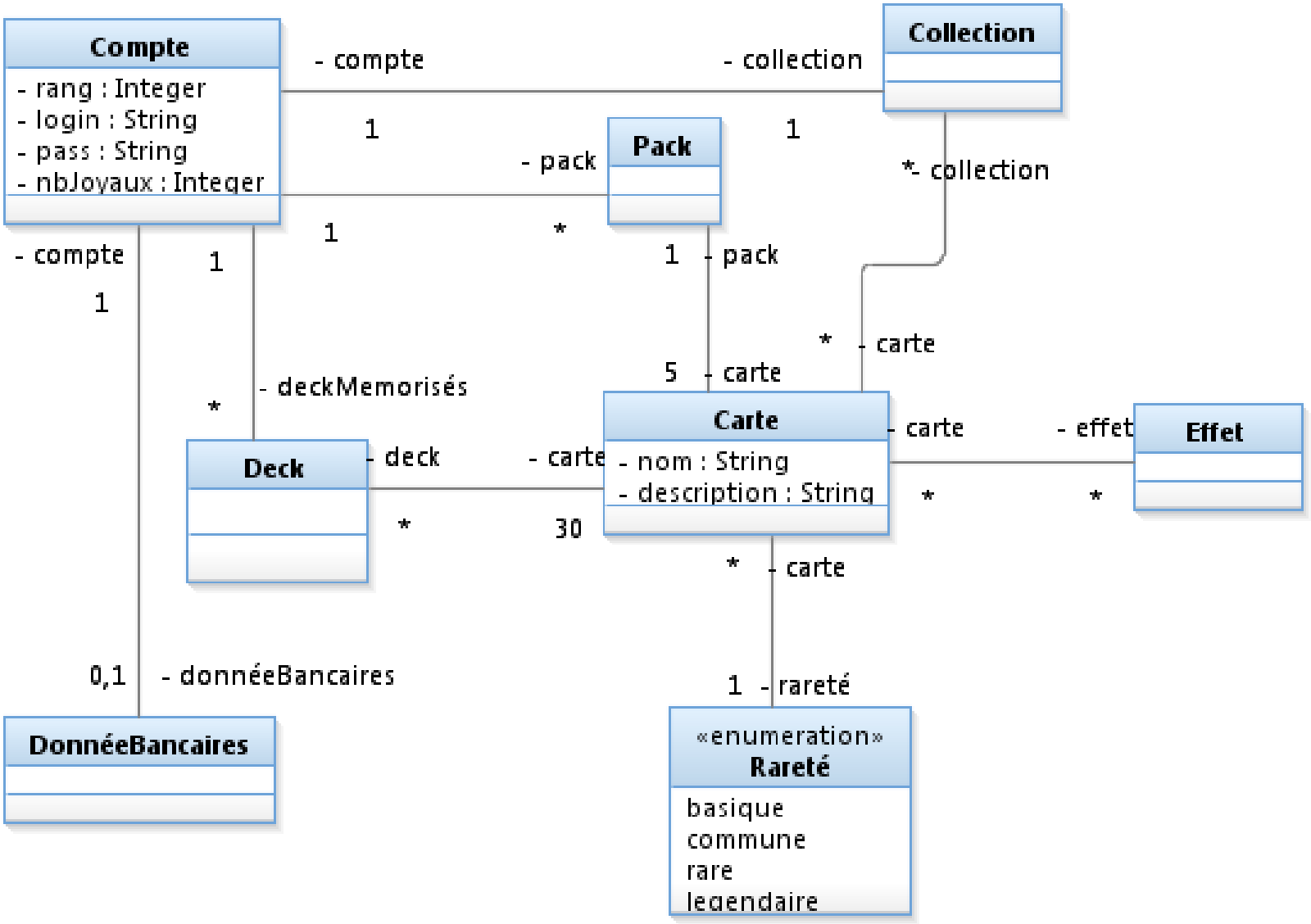
- Approche :
  - Bien lire le cahier des charges, identifier les données/propriétés citées
  - Identifier des candidats « entité », des objets persistants (eg. des objets qui ont déjà un identifiant dans le cahier des charges)
  - Regrouper les propriétés sur les classes métier (associations ou attributs)
  - S'il reste des propriétés, former des nouvelles classes métier
  - Améliorer le diagramme au fil de la rédaction des fiches détaillées :
    - les tests « Le système vérifie que [TEST] »
    - les actions « Le système enregistre/met à jour [DONNEE] ainsi» doivent être réalisables.

Quelques exemples

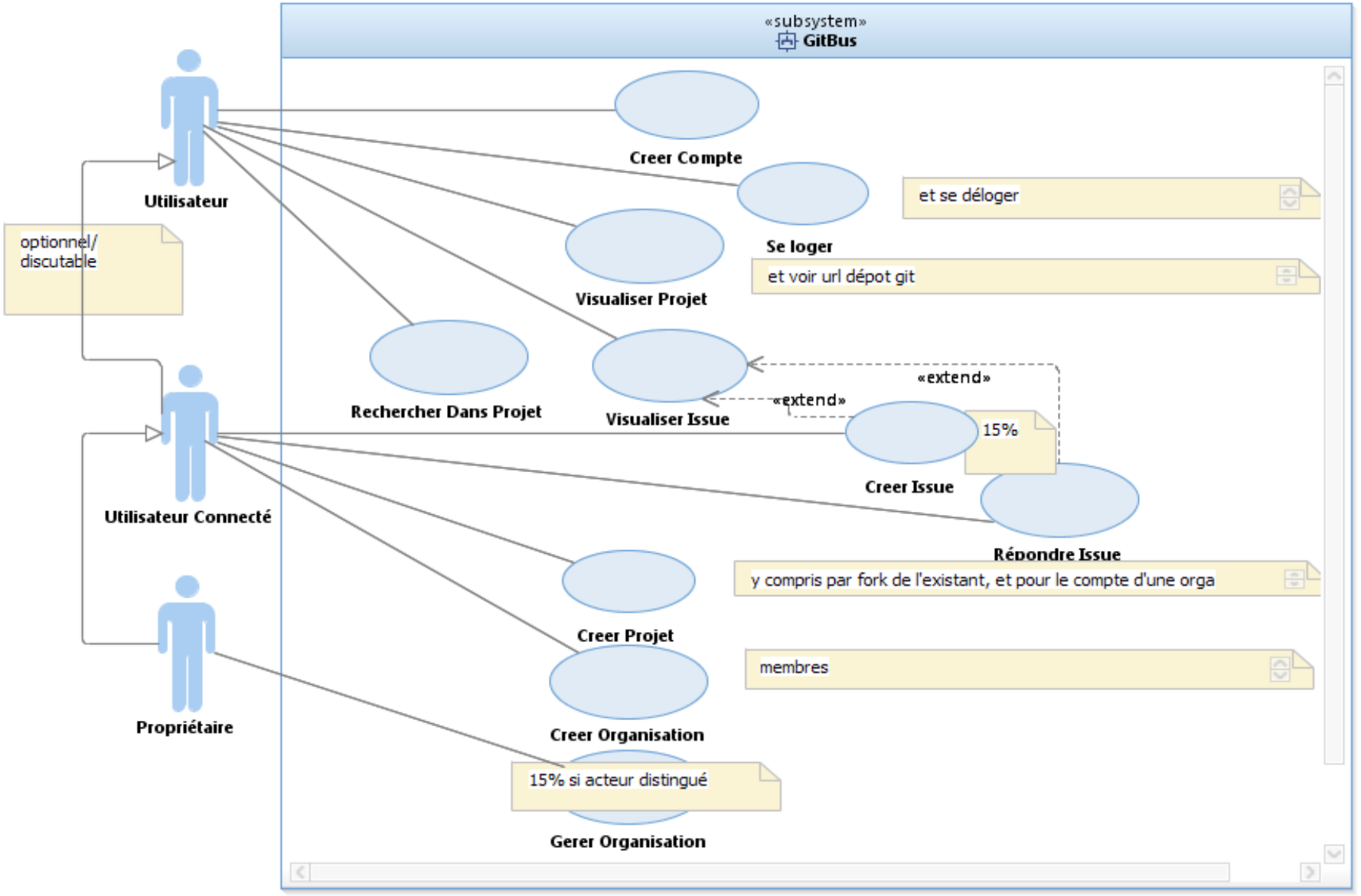
Exemple : StoneHearth



StoneHearth : Classes métier



Exemple : GitBus



GitBus : Classes métier