Doryan DENIS | 2024-2025

UML – Récap

Diagramme des Cas d'Utilisation

Composants

- **Système** : Rectangle, forme la frontière avec l'extérieur
- **Acteur** : toute entité externe au système (humains et autres systèmes comme un lancement périodique par exemple)
- **Cas d'utilisation** : actions / services proposées par le système

Acteurs toujours à l'extérieur, cas d'utilisation toujours à l'intérieur
Exemple : On ne peut pas mettre « base de données » à l'extérieur ! (Sauf si on a besoin d'appeler une BDD externe, d'un autre système)

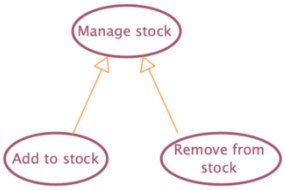
Relations

- **Acteur – Cas d'utilisation** : Association toujours. Si on a une flèche (facultatif, non conseillé) qui va de l'utilisateur vers le système alors c'est l'utilisateur qui déclenche l'action et n'attend pas forcément de retour, dans l'autre sens c'est le système qui nous envoie quelque chose sans qu'on l'ait demandé

- Entre les cas d'utilisation, 3 types de relations
  - **<<Include>>** : On a un « gros » use-case et un petit use-case, le gros fait partie du petit. Il est ici un élément obligatoire du « gros » use-case dans cette relation

  - **<<Extends>>** : Même cas de figure (« petit dans gros »), mais la « petite » opération est optionnelle cette fois

  - **Généralisation** : On a un cas d'utilisation général (A) et deux cas particuliers (B et C). A est abstrait, il est géré. En revanche B et C héritent de ce cas d'utilisation et définissent des choses plus spécifiques.
    ≠ Extends !!! Car dans extends le « petit » cas est inclus dans le « gros »



Attention ! Aucune relation ne représente la notion du temps ! C'est-à-dire que si A inclus B alors A ne se fait pas forcément avant B ! On représente les fonctionnalités en vrac dans un use-case.

Use-Case Complet / Abstrait

Dans le cas de A `----------->` B, alors A n'est pas complet (car B obligatoire)
`<<include>>`

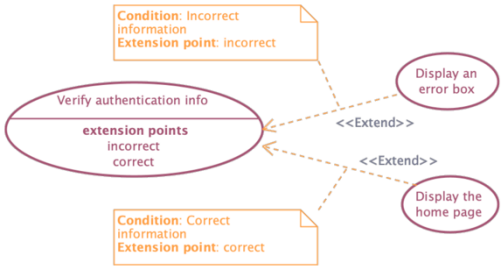
Dans le cas de A `<-----------` B, alors A est complet même si B ne se fait pas.
`<<extends>>`

La notion de complétude est pour indiquer si on peut définir le « gros » use-case tout seul.

Abstrait = on ne peut pas vraiment avoir un diagramme de séquence qui représente les étapes à l'intérieur
Exemple : « manage » n'a pas vraiment de sens, mais « ajouter » en a un
Dans le cas d'une généralisation de A, alors A n'est pas concret (il est abstrait, pas défini à lui-seul).

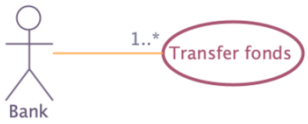
Extension Points

Non obligatoires. Représentent à quel moment on va appeler les cas optionnels, ils sont contenus dans le « gros » use-case, affichés sous la barre séparant en deux le use-case. On place un commentaire entre les relations extends pour préciser la condition et le nom de l'extension point associé.



Cardinalités

La cardinalité, placée du côté du use-case, représente le nombre d'instances du use-case qui peuvent être réalisées pour cet acteur. *Ici, la banque peut transférer 1 à plusieurs fonds (autant qu'elle veut)*. Surtout utile quand on veut dire que l'acteur ne peut déclencher qu'une fois un use-case.



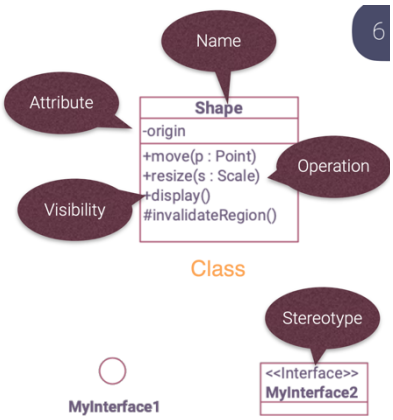
Moins fréquent, cardinalité du côté de l'acteur. Cela nous dit combien d'instances de cet acteur on doit avoir pour réaliser le use-case. *Ici, pour pouvoir jouer aux cartes on doit avoir au moins deux joueurs voire plus*.

Stéréotypes

On le trouve entre chevrons (<< >>). Ils sont utilisés pour donner un type / spécification à une relation, une classe… On ne les trouve pas dans les use-case eux-mêmes mais dans les relations qui les lient ainsi que dans le diagramme de classe. Donc pour un use-case il s'agit des relations `<<include>>` et `<<extends>>`. Dans le diagramme de classe, on trouve les stéréotypes `<<Interface>>` et `<<Abstract>>`. L'héritage n'est pas considéré comme un stéréotype !

Diagramme de Classes

Il est obligatoire avec le diagramme des cas d'utilisation.



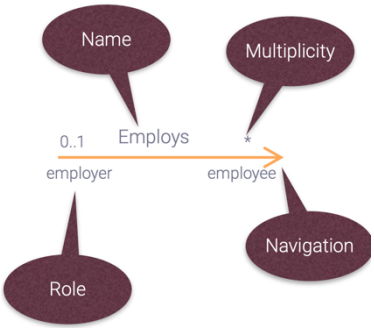
Composants d'une classe

- Nom de la classe
- Attributs
- Méthodes (Operations)
- Visibilité :
  - \+ public
  - \- private
  - # protected

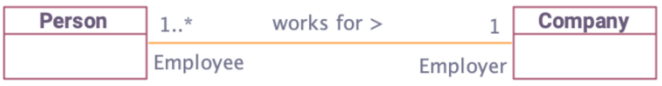
Pour les interfaces, soit on spécifie le stéréotype `<<Interface>>` au-dessus du nom sur le modèle classique de classe, soit on utilise le symbole associé (moins commun car peu aisé pour ajouter des opérations).

Relations

- **Association** : Relation assez forte entre deux classes, traduite par un attribut. Les éléments suivants sont facultatifs pour chaque association
  - Nom : Indique ce que l'association fait
  - Multiplicité : Convention relations BDD, regarder l'autre côté pour interpréter. *Par exemple, il faut lire ici : « une entreprise emploie plusieurs personnes » où entreprise est la classe à gauche, emploie = nom de l'association, plusieurs = multiplicité (côté droit !) et personnes = classe à droite. Aussi « une personne est employée par 0 ou 1 entreprise » dans l'autre sens.*

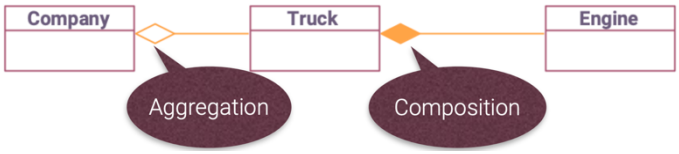


  - Rôles : Assez rare mais aident à trouver le nom des attributs que l'on trouvera dans nos classes. Surtout utile quand on a deux relations entre deux classes, désigne le rôle d'une personne selon les cas. *Par exemple ici, si la classe `Personne` est à droite, alors la classe `Entreprise` (à gauche) sera désignée par l'attribut `employer` dans `Personne`.*
  - Flèche : Navigabilité de l'objet = qui a le droit d'accéder à qui. Quand on va transformer une association en code, on va trouver les deux classes aux extrémités *(`Entreprise` et `Personne` ici)* et l'association devient un attribut qui sera dans les deux classes si la navigabilité est dans les deux sens (pas de flèche). S'il y a une flèche, alors elle précise que la navigabilité est dans un seul sens, et est dirigée vers la classe ne possédant pas d'attribut provenant de cette association. *Par exemple ici, la classe `Personne` ne contient pas d'attribut `employer`, en revanche la classe `Entreprise` aura un attribut `employee`.*
  - Flèche sur nom : On peut trouver une flèche (> ou <) au niveau du nom de l'association. C'est une indication sur le sens du verbe. *Sur cet exemple, une personne travaille pour une entreprise (et pas l'inverse).*



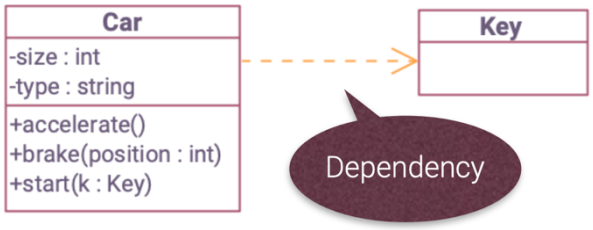
Attention ! Il est interdit d'ajouter un attribut issu d'une association dans les classes du diagramme ! Il est déjà suggéré (remplacé) par l'association.

- **Agrégation et composition** : Aussi représentées par des attributs dans les classes. Dans une association, le verbe peut être un peu ce que l'on veut ; mais dans une agrégation la relation est sémantique : c'est une association avec le nom « contient » (classe contenue dans une autre).
  Une composition est une agrégation forte : si le conteneur (classe forte) est détruit, alors le contenu l'est aussi.
  Par exemple ici, une entreprise possède des camions (ils font partie de l'entreprise) et un camion contient un moteur : les camions gardent une certaine « autonomie » par rapport à l'entreprise, mais si un camion est détruit alors son moteur l'est aussi.



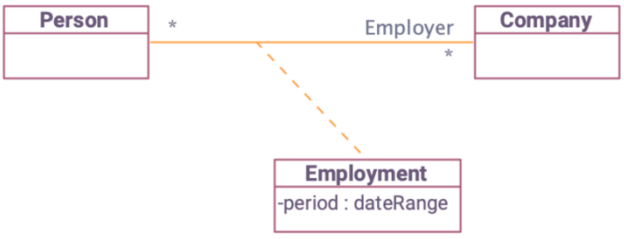
  En somme, de la force la plus faible à la plus forte des relations :
  association < agrégation < composition

- **Dépendance** : Beaucoup plus rare dans les diagrammes. Relation d'utilisation (« use »), toujours fléchée. Ça n'est pas une association particulière (≠ agrégation & composition), c'est lorsque je fais référence depuis une classe à une autre classe sans que cette dernière soit un attribut mais plutôt comme paramètre de méthode. On les représente pour voir quelles classes sont affectées si je modifie une certaine classe.

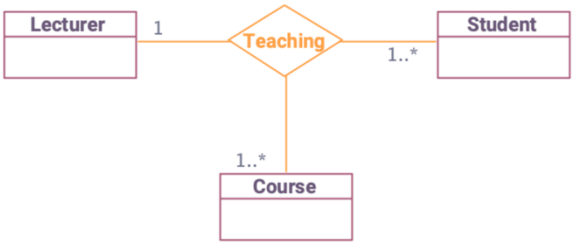


- **Généralisation** : Relation d'héritage entre classe, une sous-classe hérite des attributs et méthodes protégées et publiques de la classe mère. Les sous-classes rajoutent des informations qui leurs sont propres. Même notation que dans les use-case (flèches pleines avec triangle plein).

- **Classe d'association** : Quand on a une association mais que l'on souhaite lui rajouter des informations.
  Par exemple ici, si une personne travaille dans plusieurs entreprises, alors je pourrais avoir envie de connaître la période sur laquelle il a travaillé dans une certaine entreprise. C'est une information relative à l'association entre `Personne` et `Entreprise`, d'où l'utilisation d'une classe d'association.



- **Relation n-aire** : Très rare, on ne va généralement pas au-delà de 3. Dans une association ternaire, pour lire une cardinalité proche d'une classe on fixe 1 pour les deux autres classes. *Par exemple ici, pour lire la cardinalité près d'`Enseignant`, on se dit « pour 1 `Cours` et 1 `Étudiant` combien d'`Enseignant` pouvons-nous avoir ? ». Aussi « pour 1 `Cours` et 1 `Enseignant` combien d'`Étudiants` puis-je avoir ? au moins 1 (sinon pas un cours) et plusieurs ».*



Quelques stéréotypes supplémentaires

<<Enum>> : énumération, représente un ensemble de valeurs fixes
Controller : Classe qui s'occupe de la communication entre le modèle et la vue

Design Patterns

Définition : « Solutions communes à des problèmes communs. » Diagrammes de classe pour résoudre ces situations, définies par plusieurs experts.
3 types de design patterns :

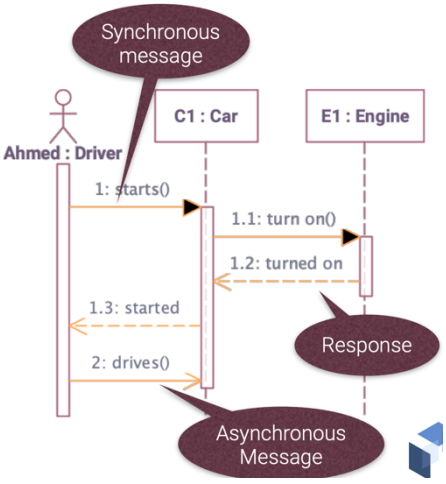
- **Structurels** : concernent la structure du code (Adapter et Composite par exemple)
- **Créationnels** : faits pour donner une solution pour crée de nouveaux objets (Factory et Singleton par exemple)
- **Comportementaux** : donnent une information sur le comportement entre les objets (Strategy et Observer par exemple)

Diagramme de Séquences

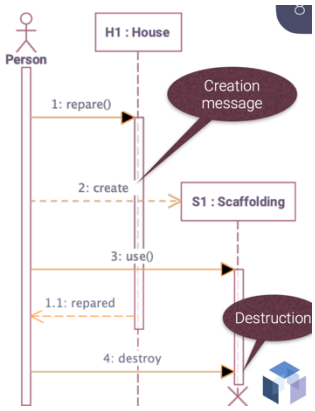
Diagramme dynamique, représente une relation temporelle d'un scénario particulier.

Composants

- **Acteurs et objets**
- **Ligne de vie** : représente la « vie » de l'objet
- **Messages** : 2 types principaux
  - Synchrones : l'objet envoyant le message est bloqué tant qu'il n'a pas reçu de réponse, représentés par des flèches à triangle plein
  - Asynchrones : l'objet envoyant le message peut faire autre chose en attendant la réponse, représentés par des flèches ouvertes
- **Réponses** : représentées par des flèches pointillées



Autres types de messages

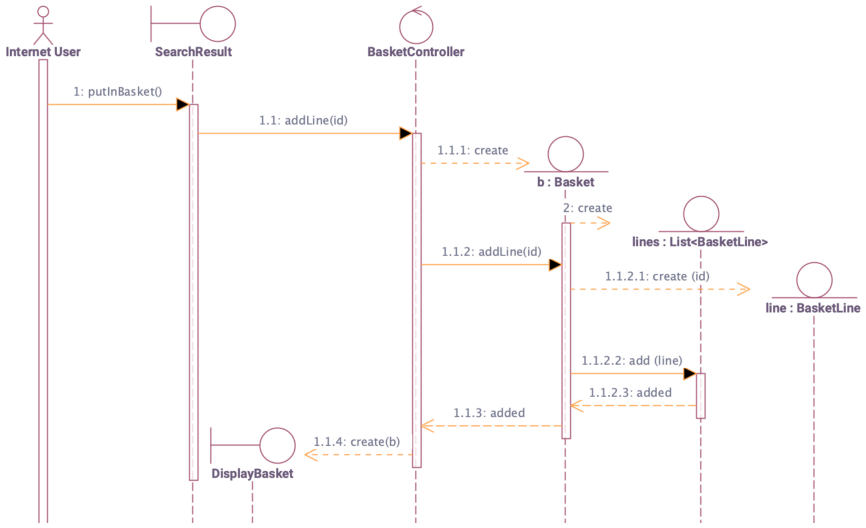


- **Création** : Message qui implique la création d'un nouvel objet (un peu l'équivalent d'un `new`), ce dernier est alors représenté plus bas que tous les autres. Cela signifie que les objets alignés tout en haut du diagramme sont là depuis le début du scénario.
- **Destruction** : Message qui implique la destruction d'un objet, la ligne de vie de cet objet sera alors interrompue avec une croix dessus au niveau de la flèche.

Icônes de stéréotypes

Plusieurs icônes peuvent être utilisées pour désigner des stéréotypes en particulier. Elles apparaissent dans l'ordre après l'acteur dans le diagramme de séquence suivant.

- **Boundary** : interface avec laquelle l'utilisateur va communiquer
- **Controller** : classe particulière permettant de faire communiquer vue et entité
- **Instance** : objet issu des classes définies dans le diagramme de classes



Attention ! Quand un objet appelle une méthode, alors cette dernière est définie dans la classe qui est au bout de la flèche. *Par exemple ici, l'acteur InternetUser déclenche la méthode putInBasket() définie dans SearchResult.*

Structures de contrôle

Représentées par un cadre autour d'une partie du diagramme avec entre crochets la condition associée dans chaque structure ou partie de structure.

- **`alt`** : Équivalent de if, on écrit une condition puis un cas else. On sépare les cas par une ligne pointillée. On peut faire un if-else en ajoutant plusieurs autres conditions avant le else final.
- **`opt`** : Représente un élément optionnel. alt est obligatoire dans le scénario (on passe forcément par là), tandis que opt ne l'est pas. On indique entre crochets le cas où cette partie du diagramme est déclenchée.
- **`ref`** : Fait référence à un autre scénario qui a été créé dans un autre diagramme.
- **`loop`** : Équivalent de `for` et `while`, cela dépend de la condition indiquée.

- Si on indique un nombre entre crochets, alors c'est une boucle `for`. Il désigne le nombre fixe de répétitions.
- Si on indique une condition (comparaison par exemple), alors il s'agit d'un `while`.

- **par** : Veut dire « parallèle », montre deux choses qui s'exécutent en même temps. On sépare ces deux actions par un trait horizontal en pointillés dans le cadre.