Les diagrammes de classes (UML)

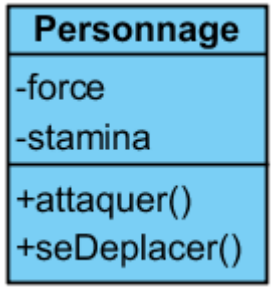
Diagramme de classes

- Décrit la **structure interne** du système, sous forme de:
  - **classes** (attributs + opérations)
  - **relations** entre les classes

Concept de classe

- Une **classe** est une description d'un ensemble d'objets qui partagent des attributs et des méthodes

- Ex: tous les personnages ont une force une stamina (les attributs) et peuvent réaliser les actions de se déplacer et attaquer (les méthodes)



Classes et objets

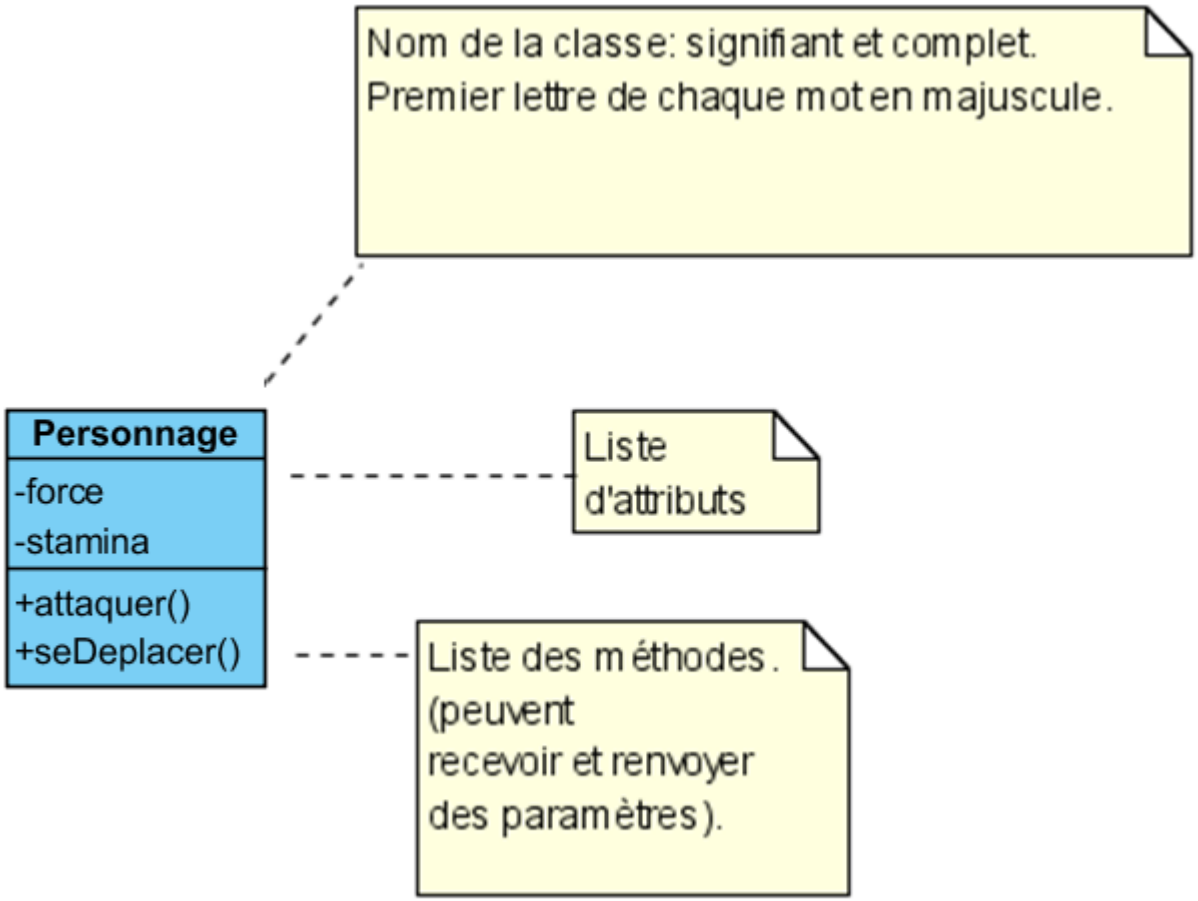
Une classe est un concept abstrait qui représente:

- des éléments concrets (les personnages, les armes)
- des éléments abstraits (un score, un de dans un jeu)
- des composants d'une application (les boutons, les images)
- des structures informatiques (cliquer un bouton, un timer)
- des éléments comportementaux (enregistrer une partie, une erreur dans un système)

Un objet (ou instance de la classe) est la **concrétisation** d'une classe

- Mario est une instance de la classe Personnage. Une arbalète est une instance de la classe Arme

Notation des classes (I)



Notation des classes (II)

- Les attributs et les propriétés peuvent avoir de différents niveaux de visibilité au niveau de code

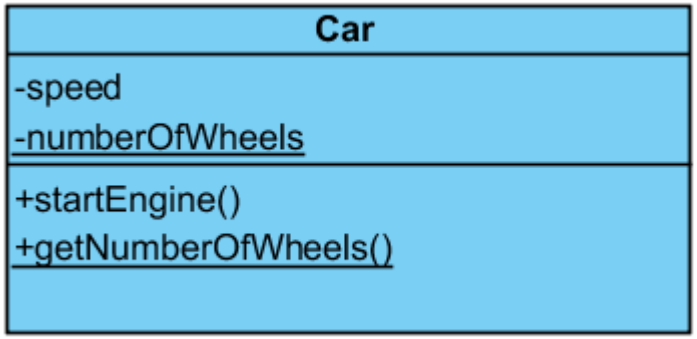
  \+ **public**: accessible en dehors de la classe, les classes filles héritent

  \- **privé**: inaccessible en dehors de la classe, pas d'héritage

  \# **protégé**: inaccessible en dehors de la classe, les classes filles héritent

- On souligne les méthodes et les attributs **de classe (static).**

Les membres statiques font partie de la classe elle-même, pas de chaque instance de la classe



Relations entre les classes

- Expriment **de liens** sémantiques ou structurels entre les objets des classes

  Ex: Une voiture est piloté par un joueur, un jeu de cartes contient plein de cartes

- Les **liens** les plus utilisées sont

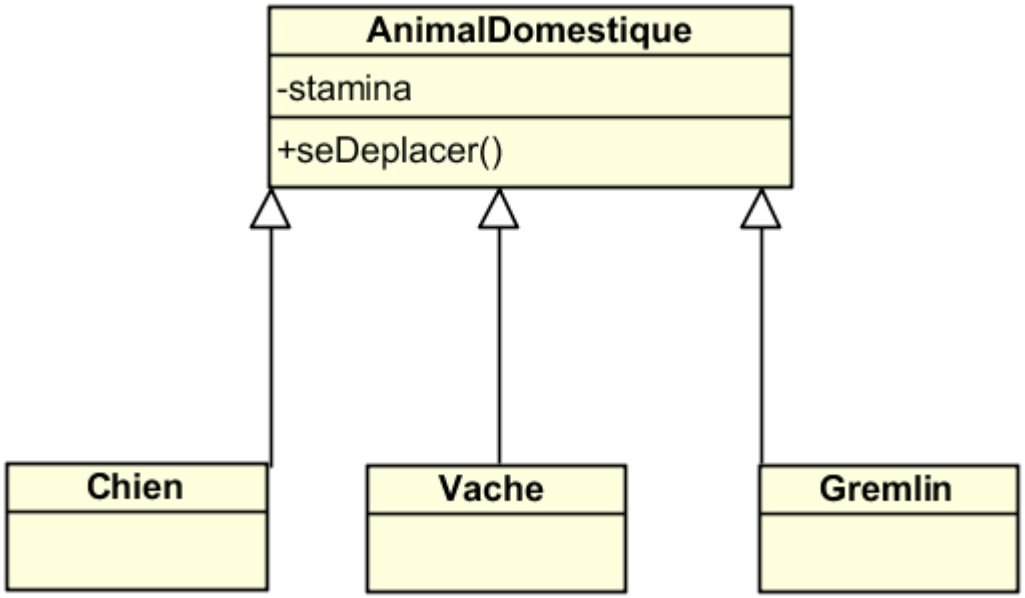
  1. l'héritage
  2. l'association
  3. l'agrégation
  4. la composition

- La **plupart** de relations sont **binaires** (deux classes liées)

- La **multiplicité** indique le nombre d'objets de chaque classe qui interviennent dans une relation entre classes

1. Héritage

- Permet à une classe d'hériter les propriétés et les méthodes non-privés d'une autre classe



Règles de l'héritage

- La classe fille possède toutes les propriétés de ses parents mais elle ne peut pas accéder à ses membres privés
- Une sous-classe peut redéfinir une méthode d'une superclasse. La définition de la sous-classe sera utilisée.
- Toutes les associations s'appliquent aux sous-classes
- On peut utiliser toujours une instance d'une sous-classe là où on utilise une instance de sa classe parent (ex: toute opération acceptant un objet Animal doit accepter un objet Chien... pas à l'inverse!)
- Une classe peut avoir plusieurs classes parents (héritage multiple... et multiplication des problèmes dans l'implémentation aussi!)

2. Association

- Le lien d'association le moins fort. **Un objet d'une classe utilise un ou plusieurs objets d'une autre classe**

- Chaque extrémité indique le rôle et la **multiplicité**

Ex: Un personnage peut manipuler d'une à trois armes et une arme peut être manipulée par plein de personnages

- Elles peuvent être "navigables" uniquement dans un seul sens

Ex: Un magicien doit connaitre le sort qu'il lance mais pas à l'inverse

La Multiplicité des Relations

- Indique le nombre d'objets (instances de chaque classe) qui interviennent dans la relation

- Les multiplicités possibles sont:

1 → 1 et un seul
* → de 0 à plusieurs
1..* → d'un à plusieurs
0..1 → de zéro à un
n..m → deux valeurs au choix (ex: 5..10 → de 5 à 10)

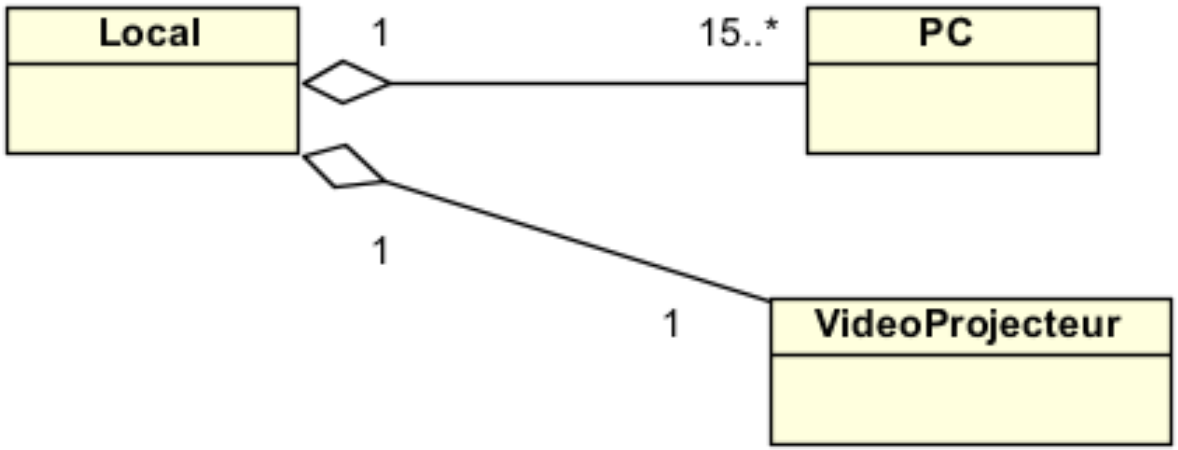
- Une association peut être modelée en utilisant un objet ou un tableau d'objets à l'intérieur des classes (ou un vecteur, arrayList, List etc…)

Exemples en C# :

```
public class Personnage {
            private List<Arme> armes;

}
public class Arme {
            private List<Personnage> personnages;

}


public class Magicien{
            private List<Sort> sorts;
}

public class Sort{
   // pas de reference vers Magicien. Un Sort ne peut pas acceder au
Magicien
   // qui le lance

}
```

3. Agrégation

- Représente l'inclusion d'un élément dans un ensemble
  - Ex: dans un local d'interface 3 il y a au moins 15 PC et un vidéoprojecteur



- La **durée de vie** des éléments qui composent l'agrégat est **indépendante** de celle de l'agrégat. Les éléments de l'agrégat (PCs, videoProjecteur) ne seront pas détruits si on détruit l'agrégat (Local)

Ex: Si on détruit le local le VideoProjecteur et les PCs existent toujours
Par contre, si on détruit exprès un PC dans le code principal, il n'existera plus dans le Local

- L'agrégation peut être modelée en utilisant un objet ou un tableau d'objets (ou une liste ou un vecteur…), selon les multiplicités

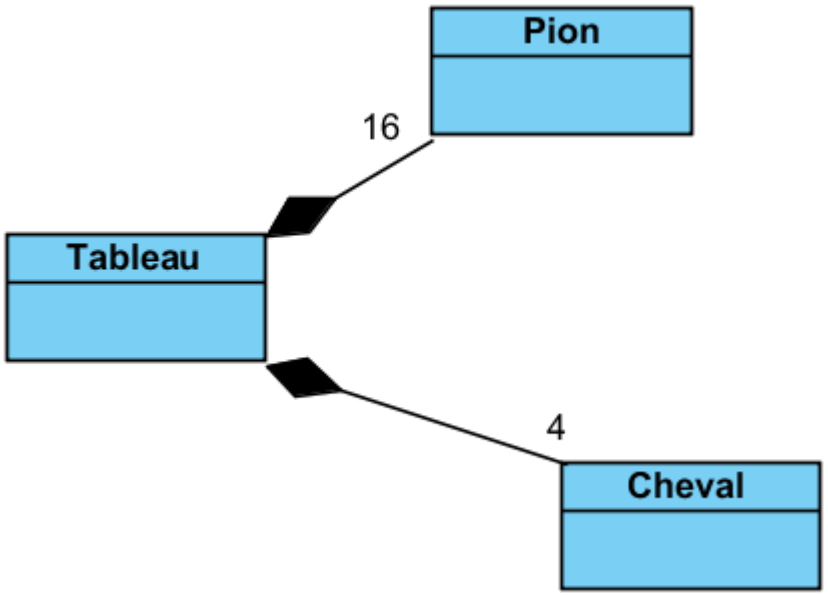
Ex: les PCs et le vidéoprojecteur existent préalablement et on les envoie au constructeur du Local. La destruction du Local n'affecte pas aux objets originaux

```
public class Local {
        private List<PC> listePC;
        private VideoProjecteur videoProj;

        public Local (VideoProjecteur videoProj, List<PC> listePC){
                this.videoProje=videoProj;
                this.listePC=listePC;
        }
}

public class PC { // code de la classe
}

public class VideoProjecteur {      // code de la classe
}
```

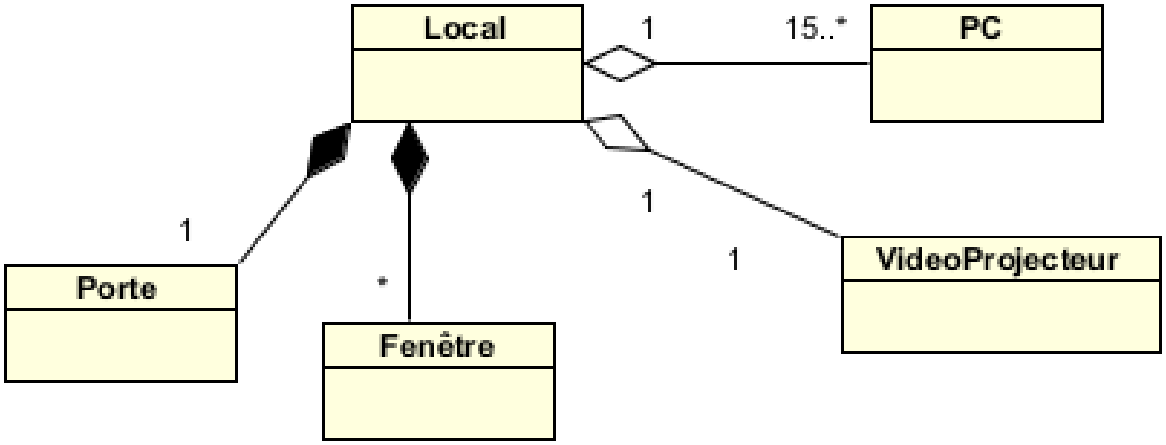
Note: Il y a plusieurs façons d'implementer une aggregation, ici on voit qu'un exemple

4. Composition

- C’est une forme particulière d'agrégation ("composite")
- Décrit une contenance structurelle entre les instances: la **création**, **copie ou destruction** de l'élément agrégat **implique la même opération** sur les éléments qui le composent
- Ex: les figures dans un tableau des échecs. Si on considère que la destruction du tableau implique la destruction des figures on obtient cet schéma



Ex: dans chaque local il y a une porte et plusieurs fenêtres qui seront détruites si on décide de "détruire" le local



- La **durée de vie** des éléments qui composent l'agrégat est **dépendante** de celle de l'agrégat (≠ Agrégation)

Ex: si on détruit l'objet Local, la Porte et les Fenêtres seront détruites aussi

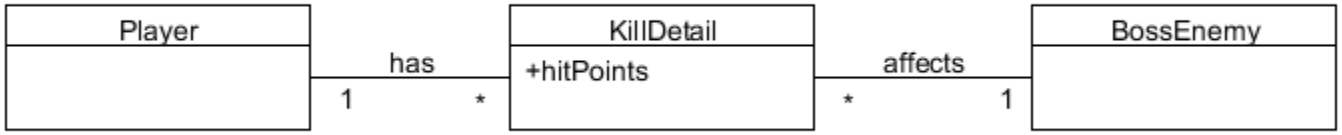
- La composition peut être modelée en utilisant un objet ou un tableau (ou équivalent: une liste, un vecteur…) d'objets. Le choix se fera selon les valeurs des multiplicités

```
public class PC { // code de la classe
}

public class VideoProjecteur {         // code de la classe
}

public class Local {

        private List<PC> listePC;
        private VideoProjecteur videoProj;
        private Porte laPorte;
        private List<Fenetre> lesFenetres;

        public Local (VideoProjecteur videoProjInit, List<PC> listePCInit){

                this.videoProj=videoProjInit;
                this.lidePC= listePCInit;
                laPorte= new Porte();
                lesFenetres= new List<Fenetre>();
        }
        public detruirePorte(){
                // détruire les objects Porte et chaque objet de
                // la liste de Fenetres pour libérer la mémoire
        }

}
```

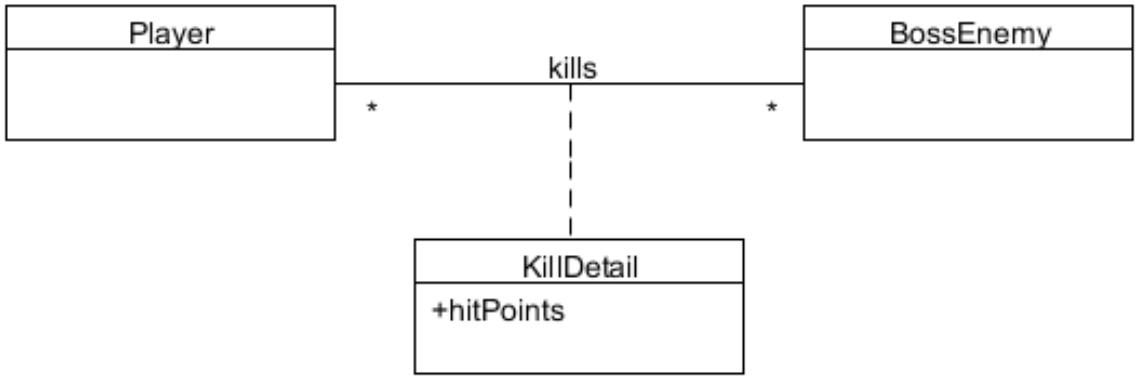
Note: Il y a plusieurs façons d'implementer une composition, ici on voit qu'un exemple

Classe Association (I)

- Quoi faire si un certain attribut concerne deux classes? Ex: le nombre de hit points que chaque joueur a fait à un boss

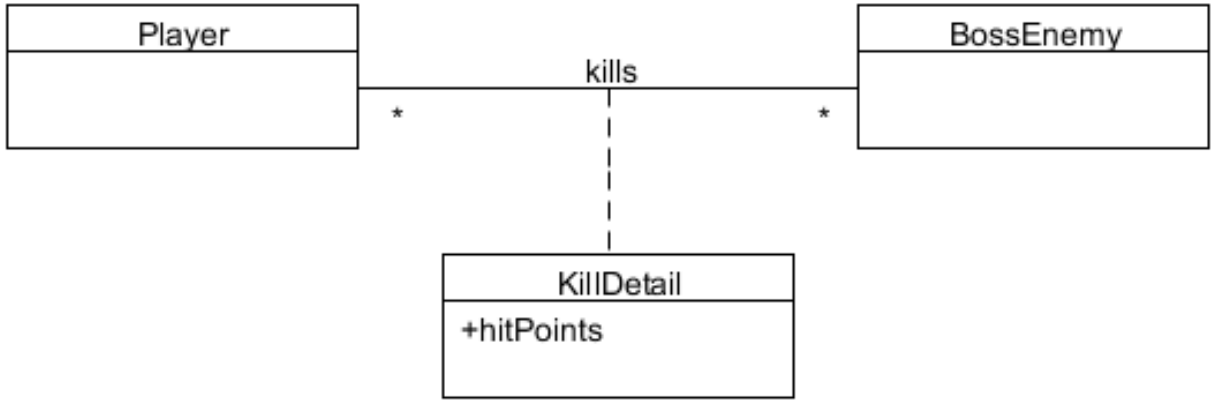


- Nous ne pouvons pas stocker cet attribut dans aucune de classes reliées car l'attribut concerne un lien entre les deux classes

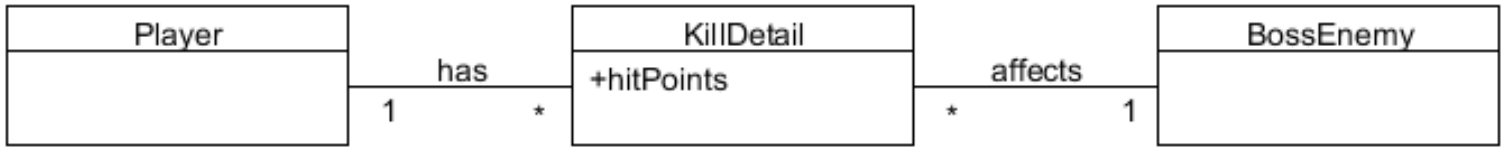


Classe Association (II)

- Il y a une autre façon de modéliser la classe association. Nous pouvons remplacer cette représentation



par celle-ci (attention aux cardinalités!)



Classe Association (III)

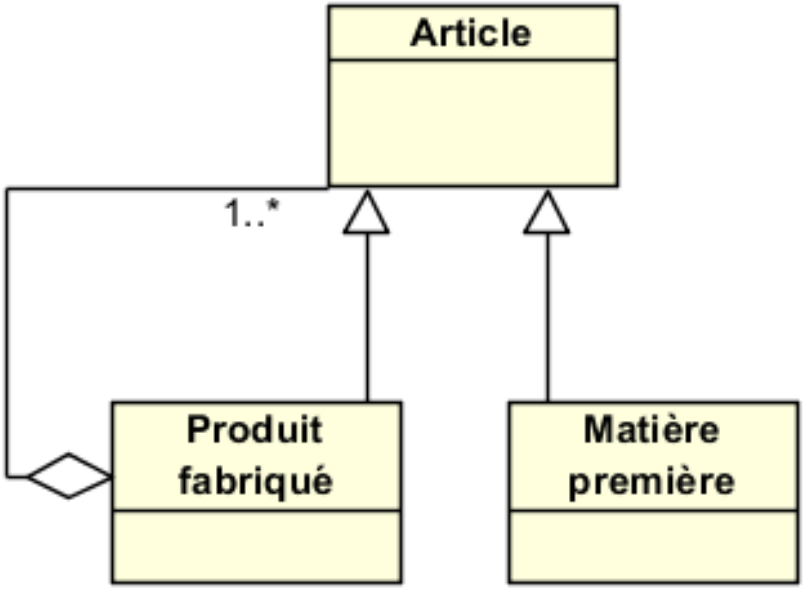
- Chaque objet de la classe association contient ses propres attributs et un objet de chaque classe associée:

```
class KillDetail {
        private Player killerPlayer;
        private BossEnemy killedBoss;
        public int hitPoints;
}

class Player{
        private KillDetail[] killDetails;
}

class BossEnemy{
        private KillDetail[] killDetails;
}
```

(les deux modèles du slide précédant génèrent le même code)

Ne pas confondre héritage et agrégation !

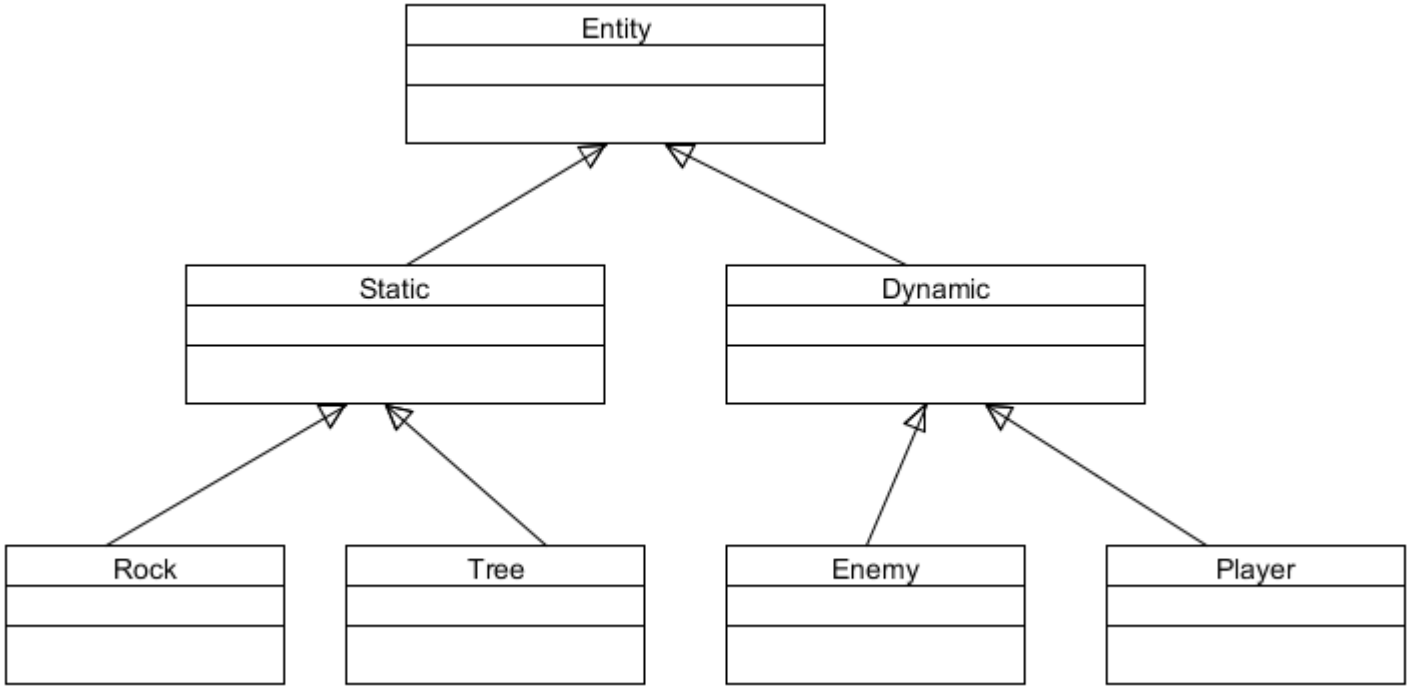
Une pomme n'est pas de la même nature qu'une tarte !



Un article est acheté (matière première) ou fabriqué à partir d'autres articles et/ou matières premières

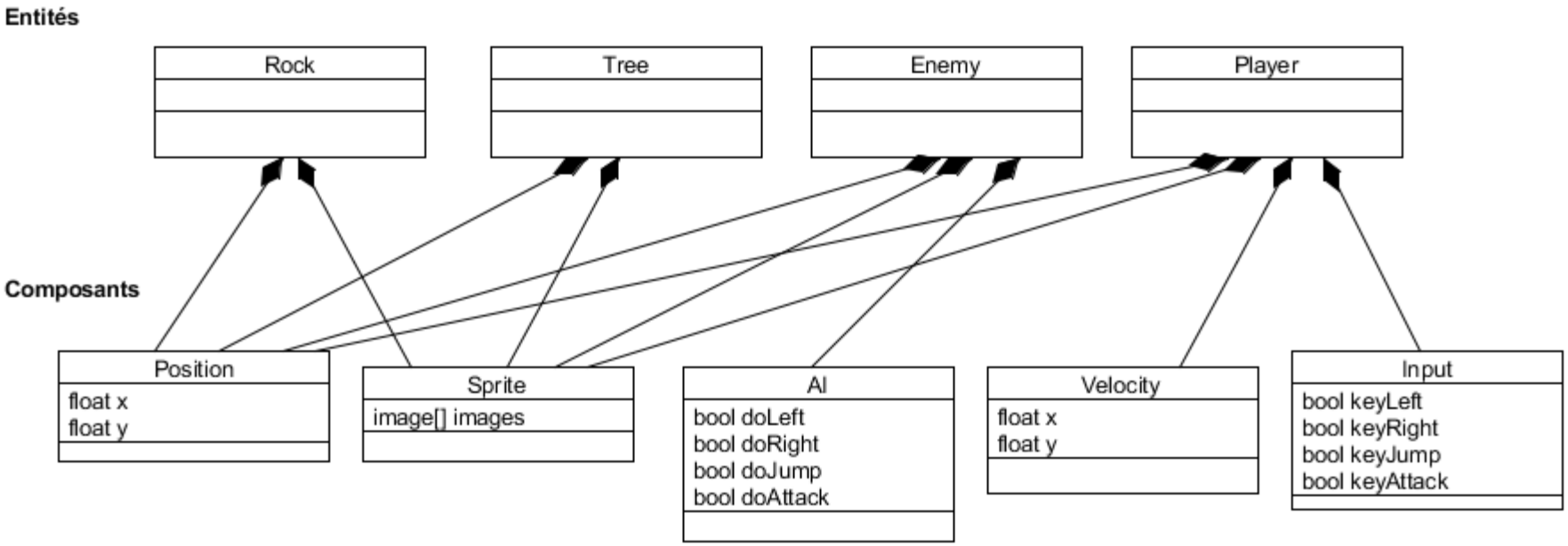
Héritage vs Agregation

- Utiliser massivement l'héritage crée une structure très rigide. Le modèle de composants basé sur l'agrégation est plus souple
- Où est-ce qu'on met ici un Tree qui soit aussi un Enemy?



Fondement du modèle de composants

- Dans le modèle de composants on construit les entités sur base d'une relation de **Composition** et pas **d'Héritage**. Les **Entités** sont de classes qui contiennent de **Composants**.



- Un composant n'a pas de méthodes, c'est juste un ensemble de données (ex: struct en C). La **logique du jeu (fonctions/méthodes)** ne **se trouve** ni dans les entités ni dans les composants, mais **dans les Systèmes, qui manipulent les composants**

- Les composants peuvent être rajoutés ou supprimés dynamiquement (impossible avec l'héritage!) car il s'agit d'une relation de composition

Mention au modèle de composants (II)

- Un **Système** est une action qui modifie l'état d'un certain groupe de composants

Ex: Considérons le système *Mouvement* d'un personnage peut opérer sur les composants *Position*, *Velocity* et *Input*

Si on veut faire sauter un personnage, ce système doit obtenir l'état de *keyJump* dans le composant *Input* de l'entité *Player*. Si c'est *true*, le système changera les valeurs du composant *Velocity* ainsi que les valeurs de *Position*

- Chaque **système** est mis à jour à chaque frame dans un ordre logique

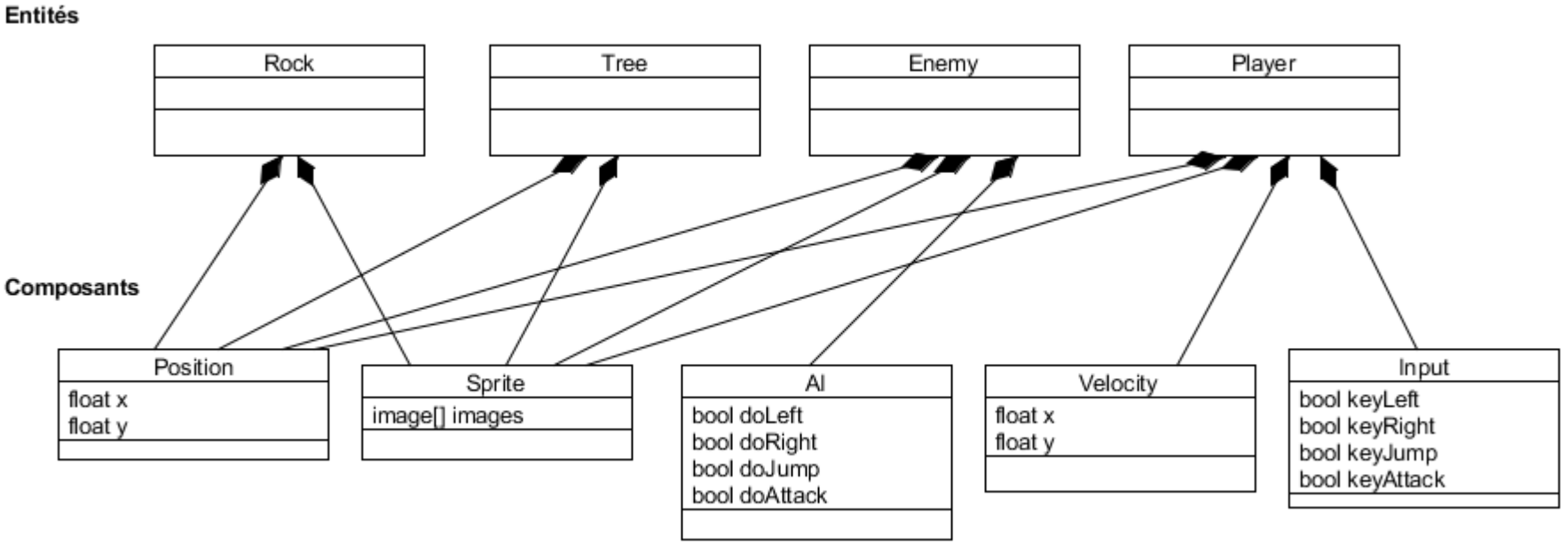
- Un système concret opère sur un ensemble de composants concret, et tous les composants doivent être présents

Ex: une entité qui a le composant *Position* mais pas *Velocity* est une entité statique. Avant on a exprimé cette situation en utilisant l'héritage de classes mais maintenant on le fait en termes de composants (ou d'absence de composants)

Mention au modèle de composants (III)

- Exemples de **Systèmes**

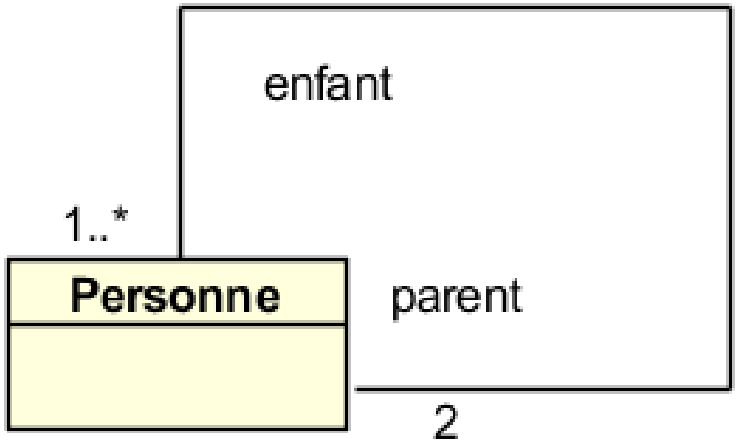
  Movement (Position, Velocity): rajouter de la vitesse à la position d'une entité
  Gravity (Velocity): accélérer la vitesse à cause de la gravité d'une entité
  PlayerControl (Input, Player): modifie l'entité *Input* d'un joueur selon un controller
  BotControl (Input, AI): Modifie les valeurs d'*Input* d'un bot en utilisant un agent *AI*
  Render (Position, Sprite): faire le rendu d'un sprite d'une entité



Associations Réflexives (I)

- Les objets d'une classe sont associés à des objets de la même classe

Ex: Créer une hiérarchie entre les objets d'une même classe



Associations Réflexives (II)

- Peuvent être modelées comme deux tableaux d'objets (un pour chaque rôle):

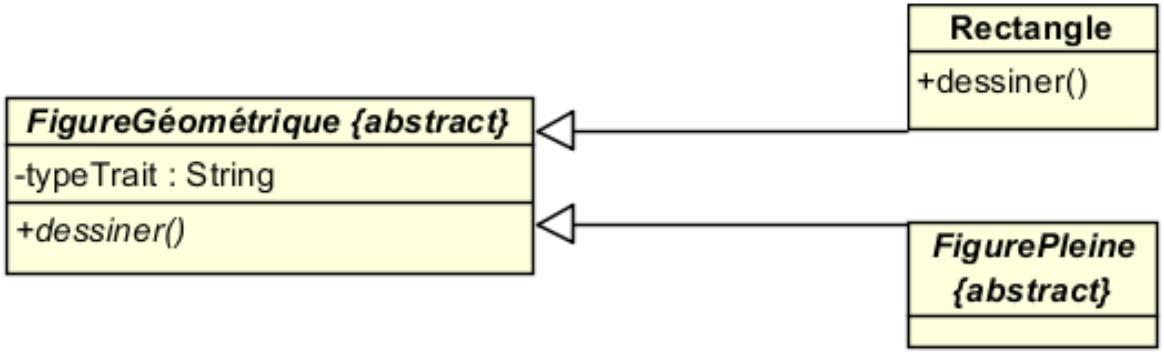
```
public class Personne {
            private Personne[] parent;

            private Personne[] enfants;
}
```

Méthodes et Classes Abstraites

- **Méthode abstraite:** Une méthode dont l'implémentation n'est pas définie. Cette méthode sera implémentée dans une sous classe

- **Classe abstraite:**

  1) si elle possède au moins une méthode abstraite (italique)
  2) si elle hérite d'une classe abstraite contenant une méthode abstraite qui n'a pas été encore réalisée



La classe *FigureGéometrique* est abstraite (1) et la clase *FigurePleine* aussi (2)

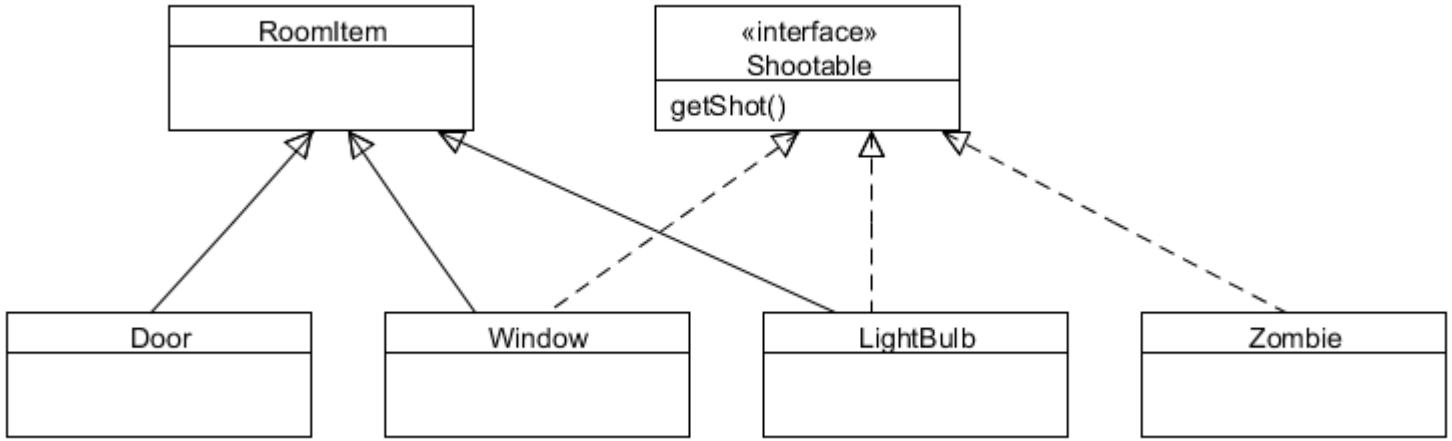
– elle hérite mais n'implémente pas la méthode. La classe *Rectangle* n'est pas abstraite: elle implémente la méthode

Remarque: On ne peut pas créer des objets d'une classe abstraite!

Interface

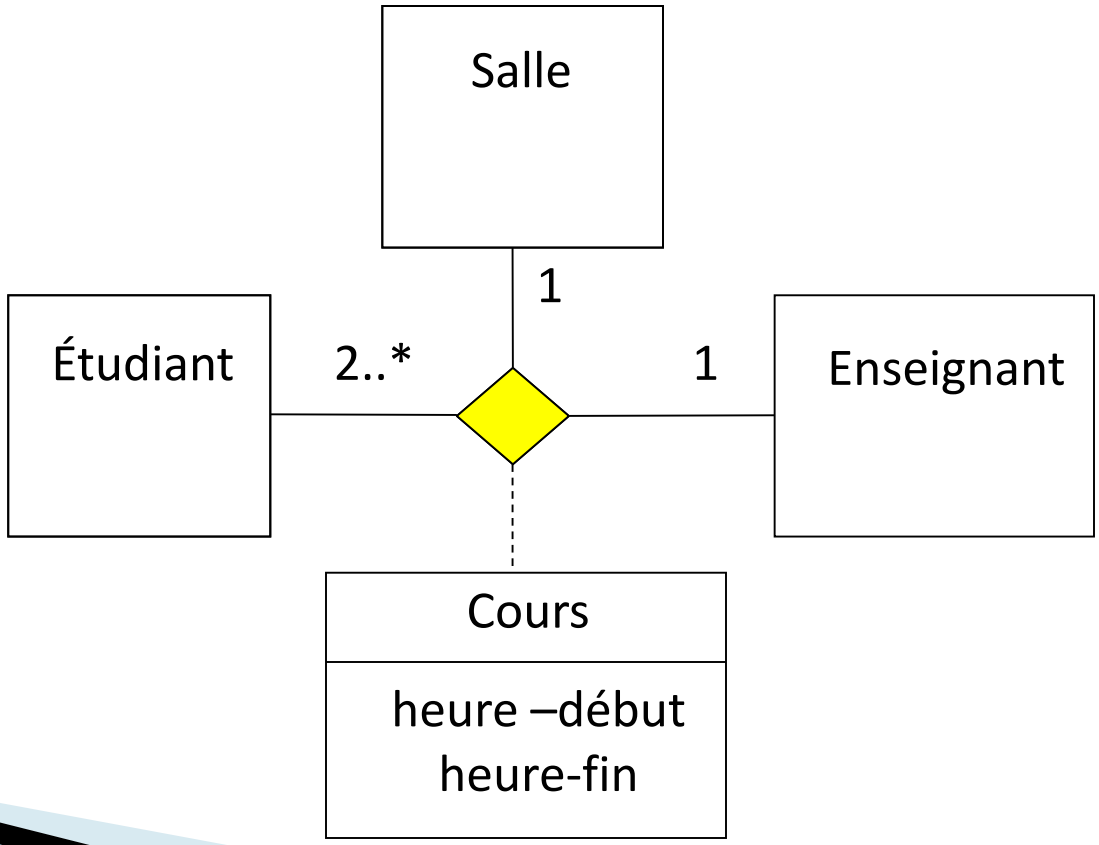
Interface: classe abstraite dont tous les méthodes sont abstraits

Une interface sert à préciser les fonctionnalités (le «service») que les classes qui l'héritent **doivent implémenter**. L'interface ne contient aucune implémentation, c'est juste un **contrat entre les classes**



Association d'arité n

Représentée par un losange avec n 'pattes' auquel peut être associé une classe porteuse d'attributs et/ou d'opérations.



Méthodologie pour la création du modèle de classes

1. Trouver les classes
2. Trouver les associations
3. Trouver les attributs de chaque classe
4. Organiser et simplifier le diagramme

Itérer et affiner le modèle!

Trouver les classes

- substantifs du domaine
  - Ex: compte, appel
  - Eviter les opérations qui s'appliquent aux objets. Ex: payer, appeler
- éliminer les classes redondantes et superflues
- éviter les noms vagues ou porteurs de rôles
  - Utilisez "Pilot" ou "Mécanicien" au lieu de "EmployéDansÉcurieDeCourseAutomobile"
  - Utilisez "Pilot", pas PremierPilote et DeuxièmePilote
- si un substantif n'est pas décomposable, il s'agit normalement d'un attribut (ex: âge)

Trouver les associations

- verbes mettant en relation plusieurs classes
  - Ex: "est composé de", "travaille pour"
- éviter les associations qui mettent en jeu plus de deux classes
  - Ex: dans l'association "conducteur-assurance-voiture", l'assurance peut être un attribut de "conducteur-assurance"
- ne pas confondre une association avec un attribut
  - Ex: l'association "est plus lourd" entre les classes Avion et Voiture peut s'exprimer avec l'attribut "masse" dans chaque classe
- éviter les chemins redondants dans les associations

Trouver les attributs des classes

- substantifs qui ne sont pas de classes
  - Ex: "la masse d'une voiture", "le montant d'une transaction"
- ses valeurs sont représentées par des adjectifs ou des expressions
  - Ex: "rouge", "50 euros"
- éviter la dépendance entre les attributs
  - Ex: si on a "datenaissance", éviter "âge"
- les attributs peuvent être ajoutés pendant toutes les étapes du cycle de vie d'un projet (implémentation comprise)