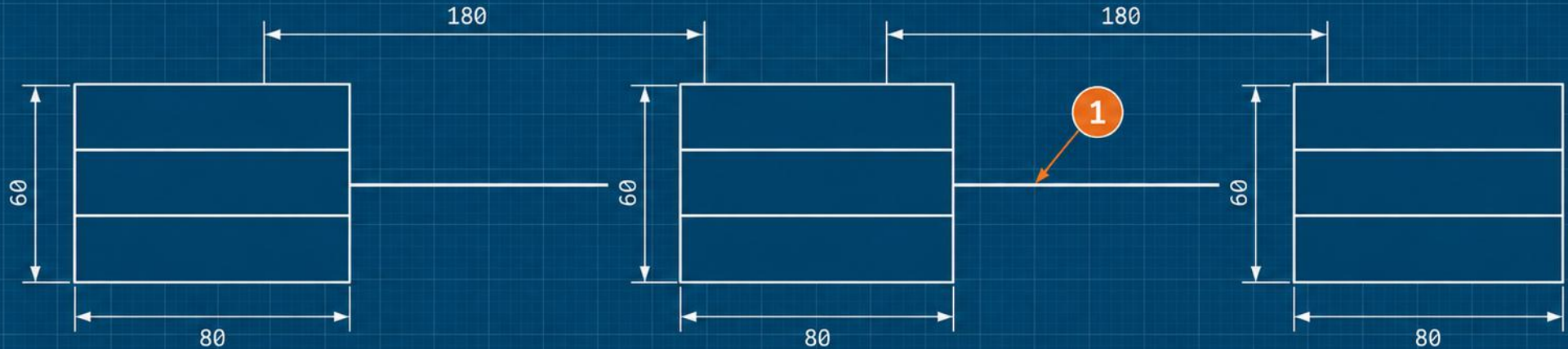


Bloc P00 · chapitre 4

Modéliser : du texte au diagramme de classes

Quelles classes écrire ? La réponse se dessine avant le code.



Combien de classes manquent ici ?

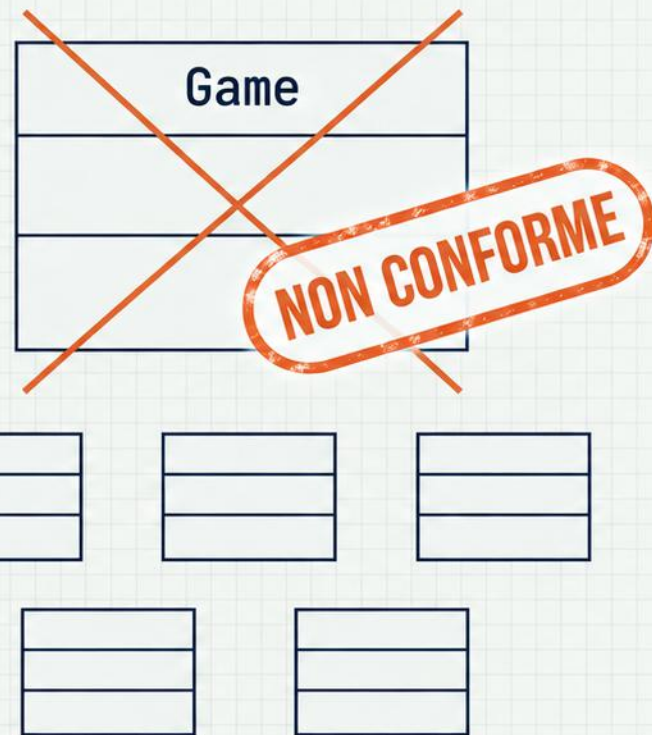
```
1 class Game
2 {
3     public array $hero = ['name' => 'Arthur', 'hp' => 10];
4     public array $rooms = [['name' => 'Entrée', 'monster' => null]];
5     public array $items = [['name' => 'Épée', 'weight' => 2.0, 'room' => 0]];
6
7     public function play(): void
8     {
9         $this->hero['hp'] -= 3;
10        $this->rooms[0]['monster'] = ['name' => 'Gobelin', 'hp' => 5];
11        echo $this->hero['name'], ' : ', $this->hero['hp'], " PV\n";
12        echo count($this->items), " objet(s) au sol\n";
13    }
14 }
```

Arthur : 7 PV
1 objet(s) au sol

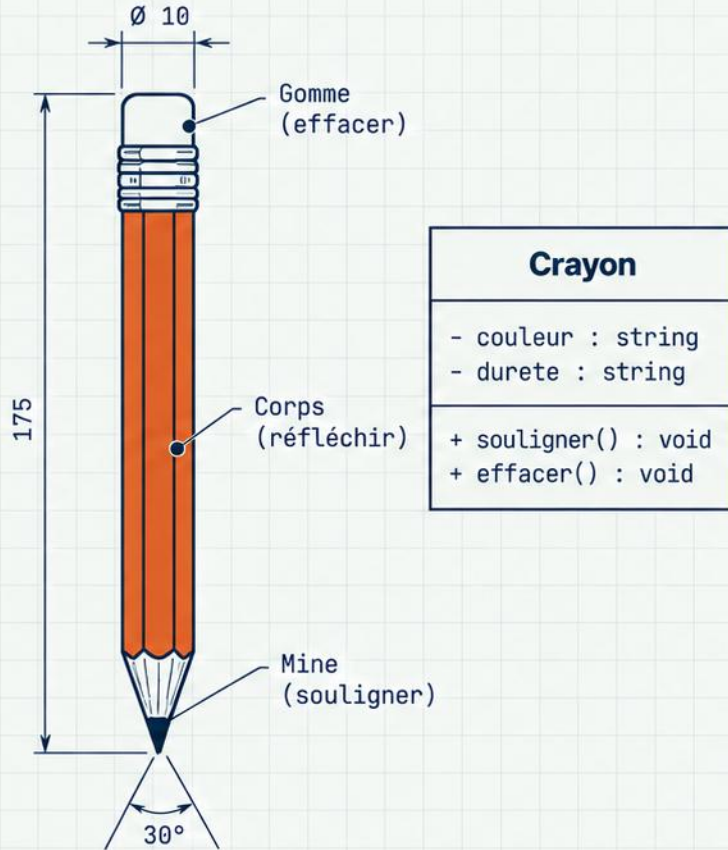
- Cinq minutes, en binôme. Ce code tourne.

Cinq classes au moins : Hero, Room, Monster, Item, Dungeon

- Chaque tableau à clés fixes ('name', 'hp') est une classe déguisée.
- Aucun type ne protège rien :
`$this->hero['hpp'] = 5` crée une case au lieu de signaler une faute de frappe.
- 'room' => 0 remplace une relation par un index de tableau.



Trois lectures de l'énoncé, un crayon



Passage	Ce que vous souseignez	Ce que ça devient
1	Les noms communs importants, ceux qui reviennent	des classes candidates
2	Les données : adjectifs, nombres, qualificatifs	des attributs
3	Les verbes : ce que le système doit savoir faire	des méthodes

L'UML dit **attribut** là où PHP dit propriété. Même chose, deux mots.

Trois questions pour trier les candidats

- **Ce nom a-t-il des données à lui ?**

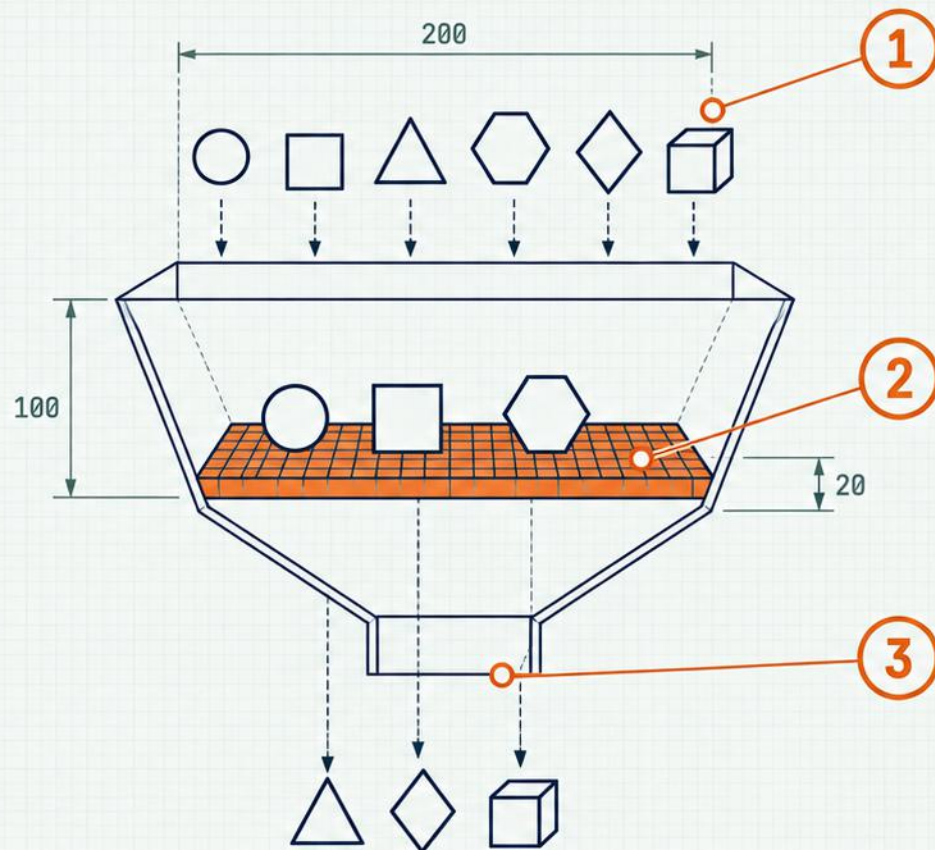
Sinon, c'est un attribut. « Le héros a un nom » :
name est une chaîne de caractères,
pas une classe Name.

- **Ce nom sert-il à autre chose qu'à l'affichage ?**

L'écran de jeu ne fait pas partie
du modèle du jeu.

- **Quelle classe détient les données de ce verbe ?**

takeDamage() modifie hp :
il va dans Hero, pas dans Game.



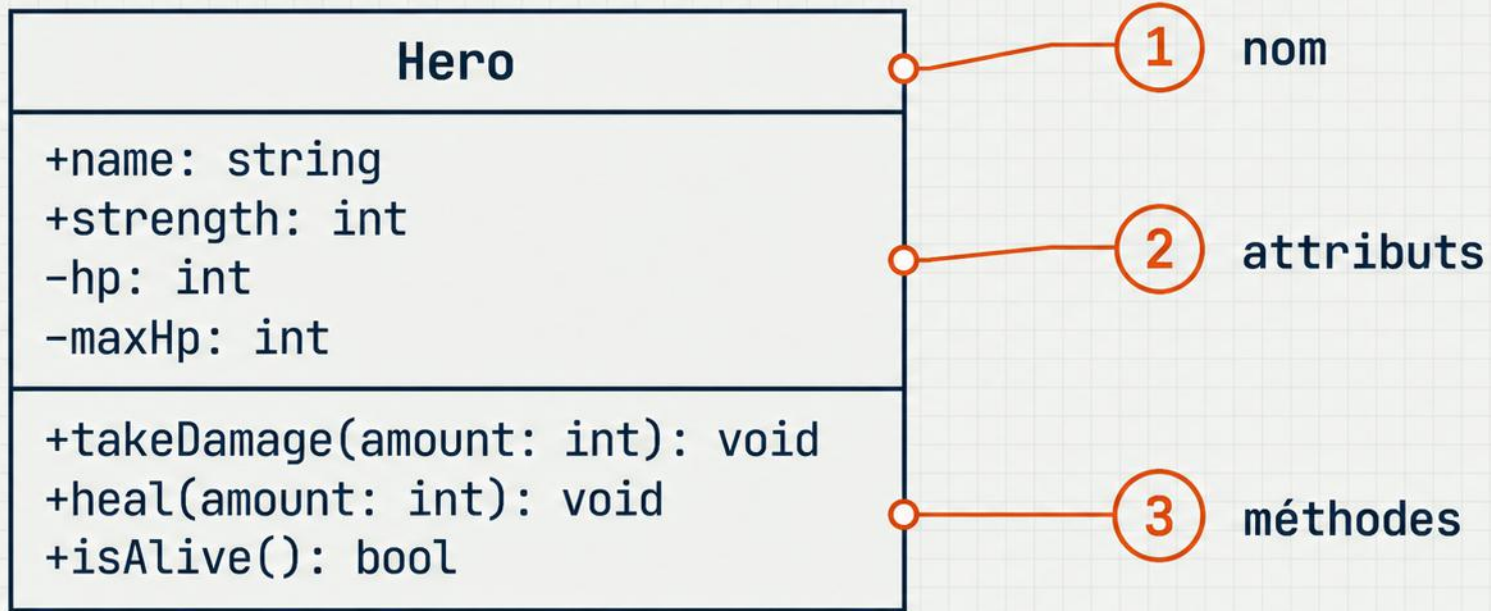
Six candidats, dont un sac qui devient une classe

« Le héros explore un donjon composé de salles ; certaines salles abritent un monstre et contiennent des objets que le héros peut ramasser dans son sac. »



- Les noms : héros, donjon, salle, monstre, objet, sac.
- Le sac a des données à lui et des actions, ajouter, retirer, peser : c'est une classe, **Inventory**.
- « Exploration » n'a aucune donnée propre : ce sera une méthode.
- « Ramasser » a besoin du sac et de l'objet : il va dans **Hero**, qui possède le sac.

Une classe : trois compartiments



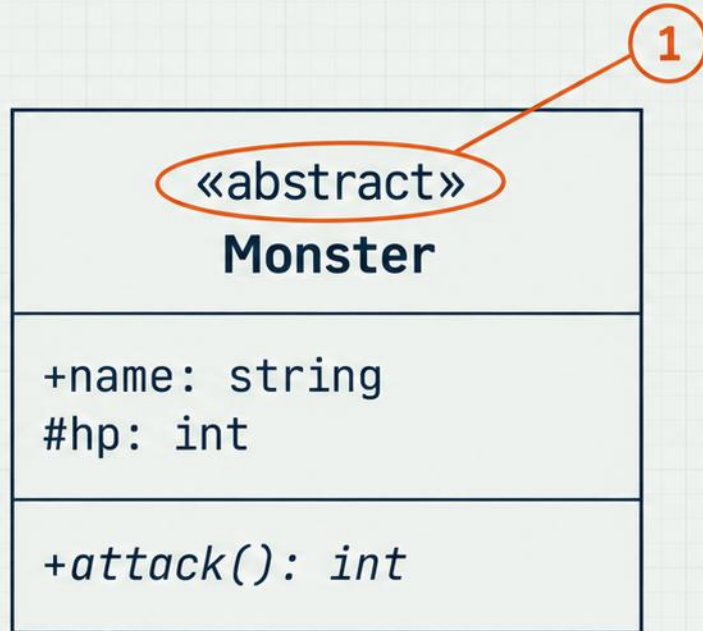
Le nom, les attributs, les méthodes. Toujours dans cet ordre.

+ - # ~ **et nom : type : tout ce qu'il y a à connaître**

Symbole	Signification
+	public : visible de partout
-	private : visible seulement dans la classe
#	protected : la classe et ses classes filles, au chapitre 6
~	visible dans le même paquet. PHP n'a pas cette visibilité
nom: type [0..1]	un attribut facultatif, qui peut valoir null
souligné	membre statique. PlantUML le note {static}

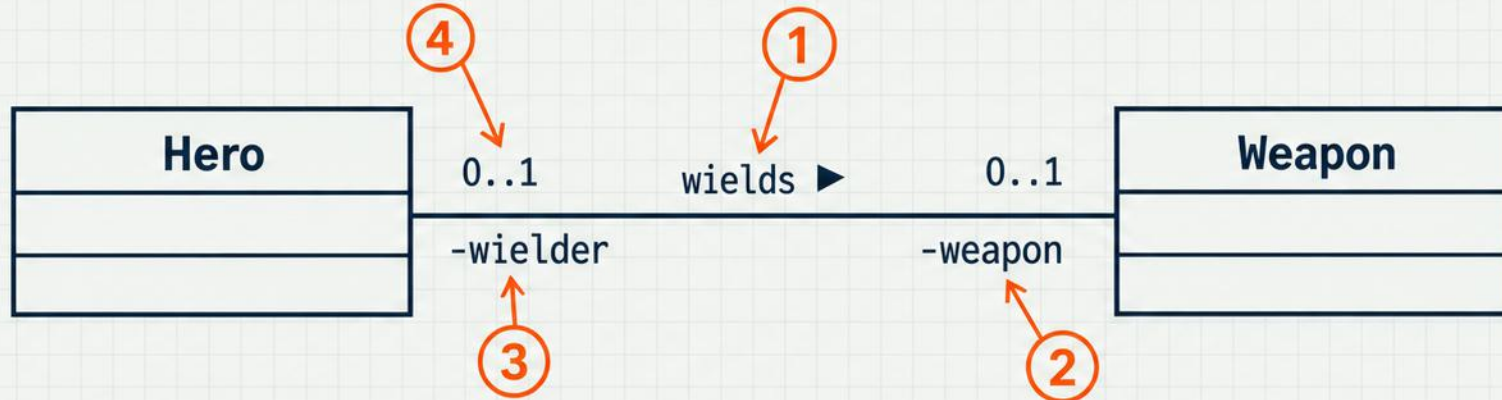
Convention du cours : une propriété public private(set) se note -, seule la classe l'écrit.

«abstract» : une classe dont on ne fabrique jamais d'objet directement



- Deux notations existent : le nom en italique, ou le stéréotype «*abstract*» au-dessus du nom.
- Le cours recommande le stéréotype : il se lit sans hésiter, au tableau comme à l'écran.

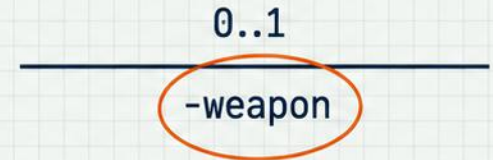
Une association porte quatre choses



1. Un nom, un verbe, et le triangle donne le sens de lecture.
2. Deux rôles, un à chaque bout : le rôle dit comment la classe d'en face appelle celle-ci. Côté Weapon, weapon est la propriété \$weapon de Hero.
3. Une visibilité par rôle, -, + ou #. Un rôle privé devient une propriété privée.
4. Deux multiplicités, une à chaque bout : combien d'objets participent au lien.

Un rôle par côté, sans exception

Multiplicité	Se lit	En PHP
1	exactement un	une propriété obligatoire
0..1	zéro ou un	une propriété qui peut valoir null
* ou 0..*	zéro ou plusieurs	un tableau, éventuellement vide
1..*	au moins un	un tableau jamais vide

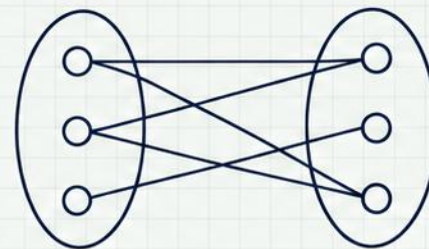
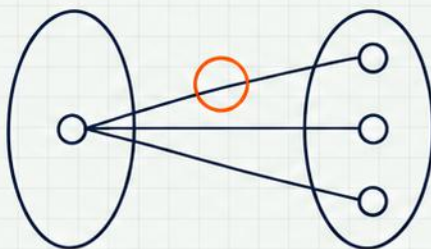
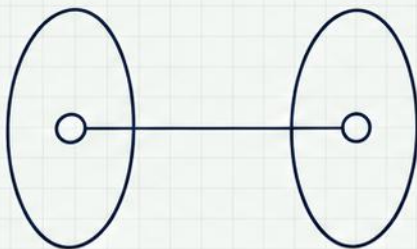


Agrégation et composition comprises. Dans le code, vous n'êtes pas obligé de programmer les deux sens du lien. Le modèle, lui, nomme les deux.

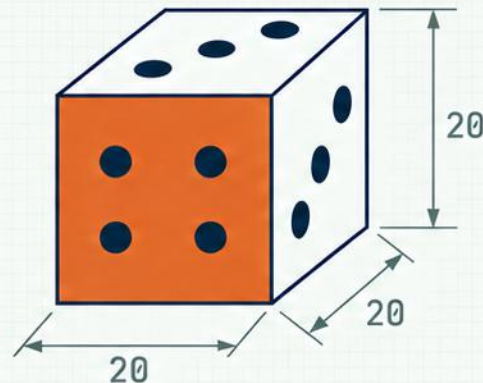
Un à un, un à plusieurs, plusieurs à plusieurs : les mêmes qu'en Laravel

Multiplicités	Exemple du Donjon	Dans Laravel
1 — 1	Hero porte Inventory	hasOne d'un côté, belongsTo de l'autre
1 — 0..*	Dungeon contient Room	hasMany du côté 1, belongsTo du côté plusieurs
0..* — 0..*	Hero accomplit Quest	belongsToMany des deux côtés, avec une table pivot

Un à plusieurs est la relation la plus fréquente : un Post a plusieurs Comment, un Comment appartient à un seul Post.



Dépendance : « utilise », le temps d'un appel



- Un trait pointillé fléché. Une classe a besoin d'une autre le temps d'un appel, sans la garder.
- Une méthode de Hero reçoit un Dice en paramètre pour lancer un jet, puis l'oublie : le dé n'est rangé dans aucune propriété.
- C'est la relation la plus faible du lot. Dessinez-la seulement si elle aide à comprendre la méthode.

Le losange est du côté du tout, et le cycle de vie décide

Agrégation



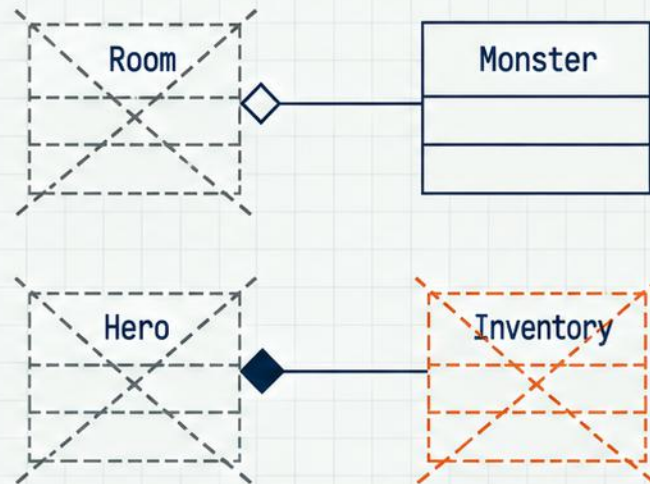
Composition



- 1 Agrégation, losange vide, « a un »** : le tout et la partie ont des vies indépendantes. Le gobelin existait avant la salle, et il lui survit.
- 2 Composition, losange plein, « est fait de »** : la partie est créée par le tout et meurt avec lui. Multiplicité 1 côté tout, une seule composition à la fois.

La réponse est déjà dans le losange

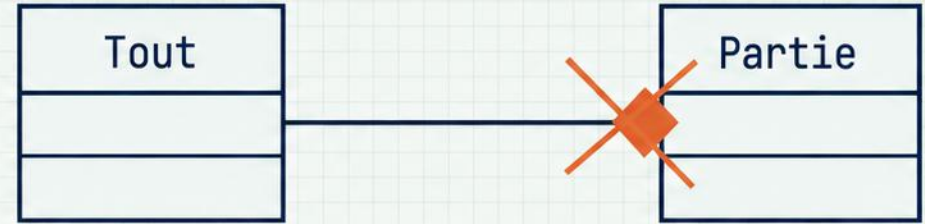
Losange	Quand le tout est supprimé	Clé étrangère
vide, agrégation	la partie reste, son lien est vidé	ON DELETE SET NULL
plein, composition	la partie est supprimée avec lui	ON DELETE CASCADE



Le test qui tranche : si je supprime le tout, la partie a-t-elle encore un sens toute seule ? Oui, agrégation. Non, composition.

Beaucoup de supports intervertissent les deux définitions

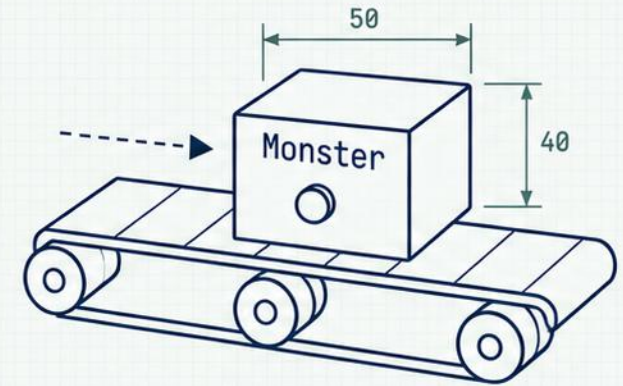
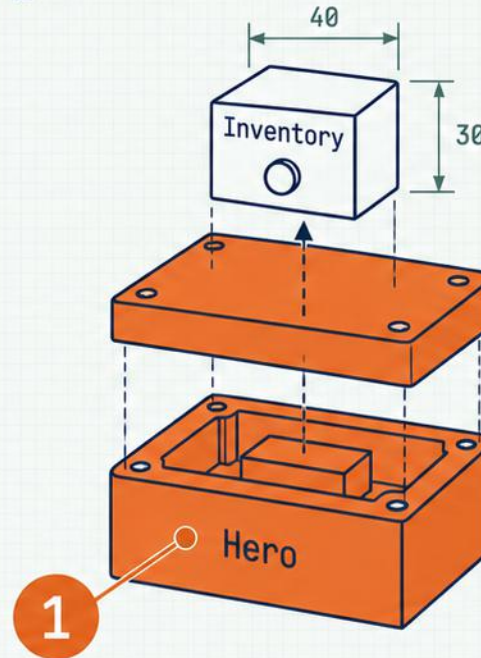
- L'agrégation n'est pas le lien fort. C'est l'inverse.
- Losange vide : la partie survit au tout.
- Losange plein : la partie meurt avec le tout.



NON CONFORME

Un indice fiable la plupart du temps : qui fabrique l'objet ?

- Si le tout fabrique la partie dans son constructeur, c'est une composition.
- Si la partie arrive de l'extérieur, c'est une agrégation ou une association.
- Cet indice se trompe parfois. Quand il contredit la phrase « est fait de », suivez la phrase.



Les quatre relations de ce chapitre, et les deux suivantes

Relation		Se lit	Durée de vie de la partie	En PlantUML
Association		« est en lien avec »	indépendante	A -- B
Dépendance		« utilise »	le temps d'un appel	A ..> B
Agrégation, losange vide		« a un »	survit au tout	A o-- B
Composition, losange plein		« est fait de »	meurt avec le tout	A *-- B

Chapitre 6, la généralisation, triangle vide, « est un ». Chapitre 7, la réalisation, triangle vide en pointillé, « respecte le contrat ».

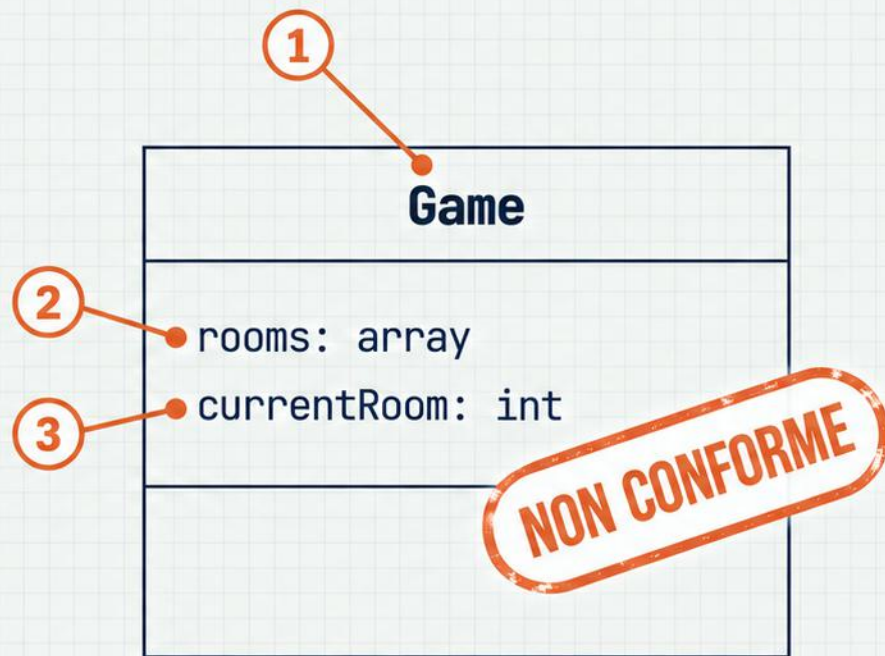
Qu'est-ce qui cloche dans ce modèle ?

Game
-heroName: string -heroHp: int -rooms: array -items: array -monsters: array -currentRoom: int
+play(): void +move(direction: string): void +fight(): void +pickUp(itemName: string): void +save(): void +printScreen(): void

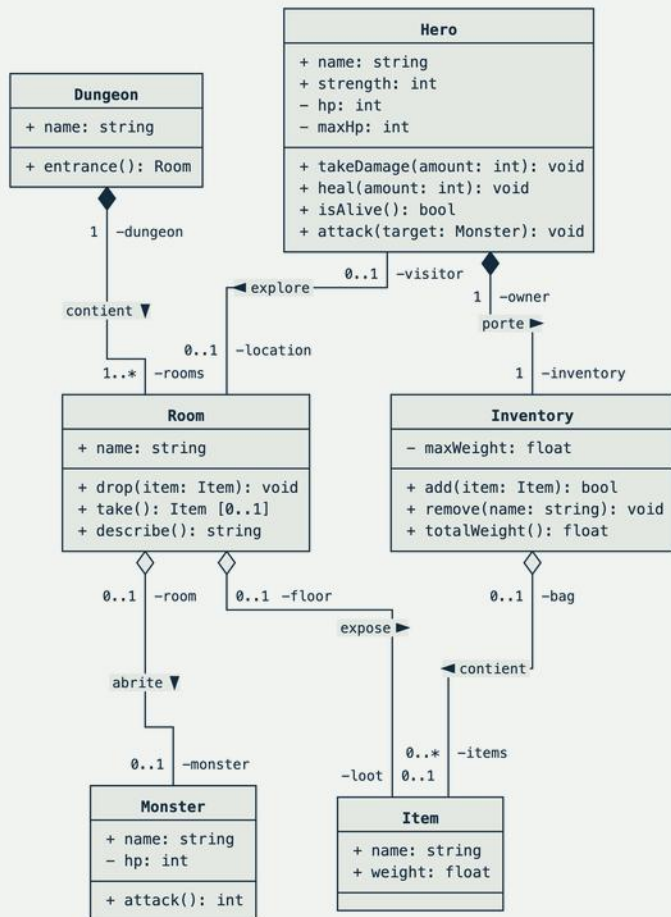
Trois minutes. Listez ce qui cloche avant la slide suivante.

La classe qui fait tout, les tableaux, les index

- **La classe qui fait tout.** Game sait tout et fait tout. Vous ne pouvez ni la tester par morceaux ni travailler dessus à deux.
- **Les attributs en tableaux:** rooms: array ne dit rien de son contenu. Un tableau à clés fixes est presque toujours une classe qui n'a pas été écrite.
- **Les relations remplacées par des index.** currentRoom: int désigne une case du tableau rooms. Si l'ordre du tableau change, tout casse.



Lisez le diagramme en français

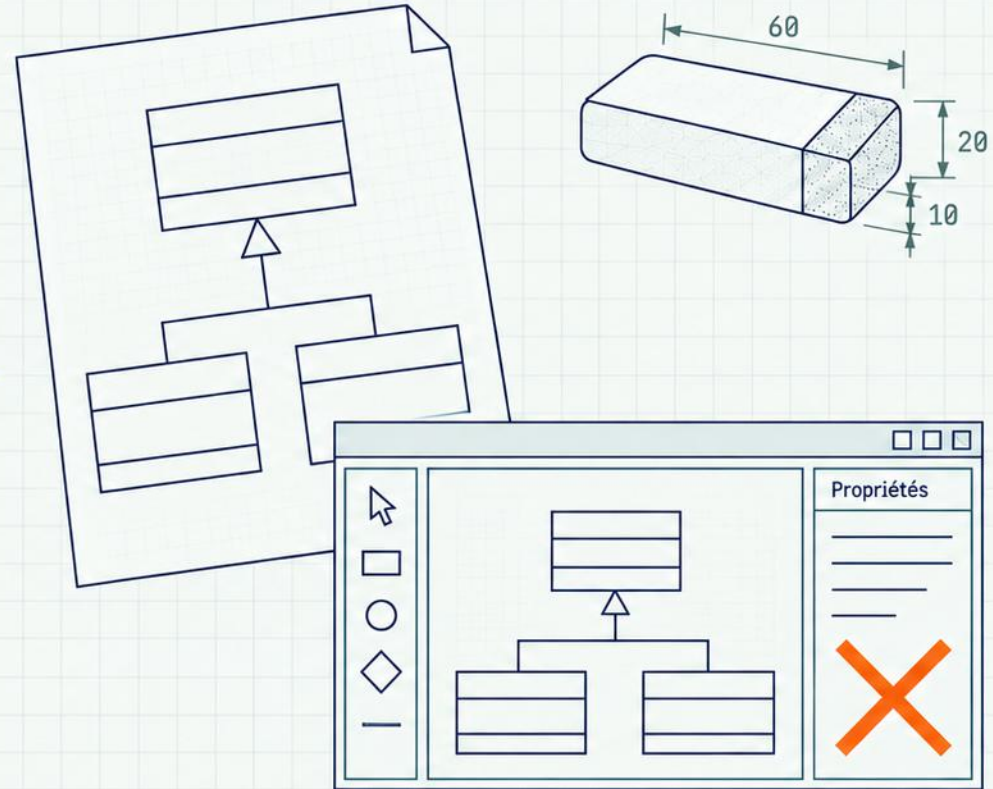


« Un donjon **est fait** d'au moins une salle ; une salle **a** au plus un monstre et **a** au plus un objet au sol ; un héros **est fait** d'un inventaire ; un inventaire **a** zéro ou plusieurs objets ; un héros explore au plus une salle à la fois. »

- `currentRoom: int` a disparu, remplacé par l'association `explore` entre `Hero` et `Room`.
- Le combat n'est pas un trait : `attack(target: Monster)` utilise le monstre le temps d'un appel.

Papier et crayon d'abord, un outil ensuite

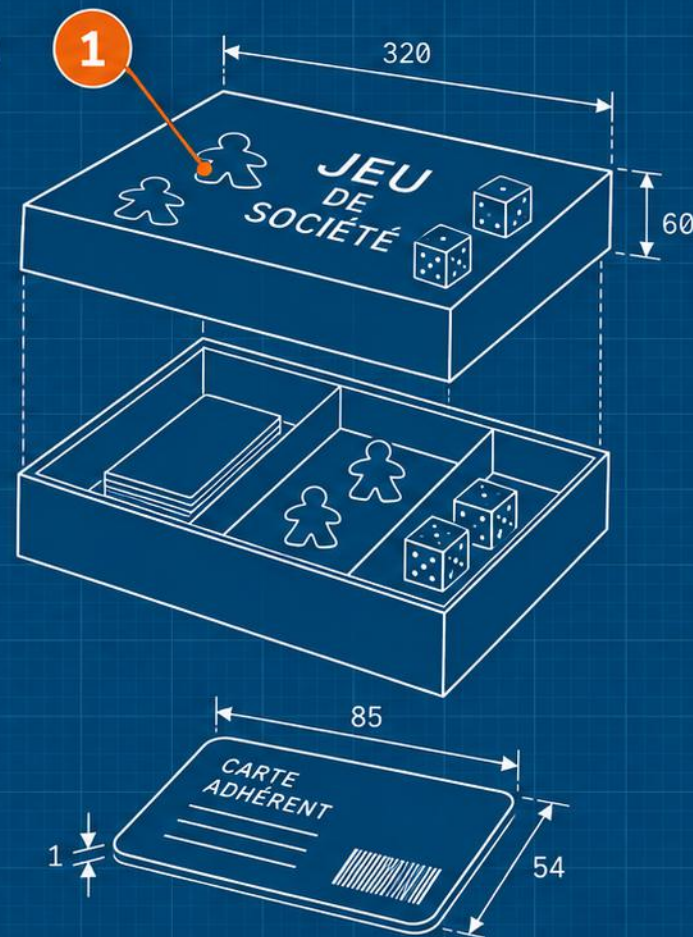
- Une feuille, une gomme, des boîtes vides, puis les traits. Vous effacez un losange en deux secondes.
- **Modelio**, recommandé : interface graphique, diagrammes conformes à UML 2.
- **StarUML** si Modelio ne s'installe pas chez vous. **PlantUML** en texte, pour versionner le diagramme avec le code.
- **Mermaid est interdit** : il ne respecte pas le standard UML 2. Un diagramme Mermaid sera refusé.



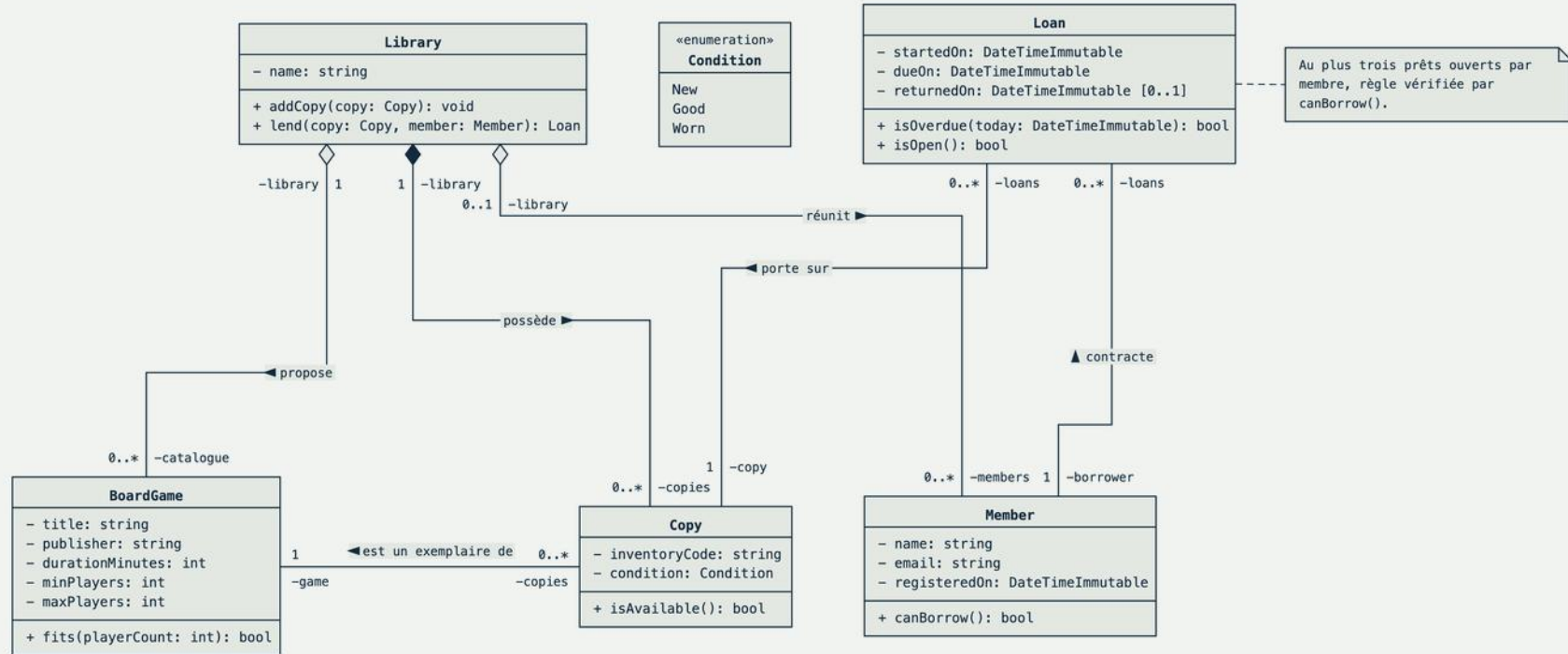
Exercice : la ludothèque

Rendu attendu : **un diagramme de classes** avec les attributs typés, les méthodes publiques utiles, les relations avec **leurs deux rôles** et **toutes** les multiplicités. Pas de code. Papier d'abord, mise au propre ensuite. Comptez 40 minutes.

1. Combien de classes ? Comptez-les avant de dessiner.
2. Où sont les compositions ? Cherchez dans l'énoncé les parties qui disparaissent avec le tout.
3. Le prêt est-il une classe ou un simple trait ? Regardez s'il a des données à lui.
4. Comment traduisez-vous « jusqu'à trois exemplaires à la fois » ?
5. Où mettez-vous la méthode qui dit si un prêt est en retard ?



Corrigé : cinq classes et une énumération



- **Loan est une classe.** Dès qu'un lien porte des données, il devient une classe.
- **Library *-- Copy est une composition, Library o-- Member une agrégation :** les membres existaient avant.
- **« Trois exemplaires à la fois » n'est pas une multiplicité.** La règle vit dans `canBorrow()`.