POO et UML-2
Introduction - suite
Codes objets en Java et Python : 2
Diagrammes de classes associés
Principes de POO - GRASP

Bertrand LIAUDET

SOMMAIRE

Edition février 2025

1 - ASSOCIATIONS ENTRE LES CLASSES

0 – Références Java et Python

- La présentation est associée à un ensemble d'exemples de code téléchargeables en Java et en Python.

Sources Java :

https://docs.xoracle.com/javase/9/docs/api/index.html?overview-summary.html : en mode frame

Sources Python :

https://docs.python.org/fr/3/

https://docs.python.org/fr/3/tutorial/classes.html

https://www.courspython.com/

https://www.courspython.com/classes-et-objets.html

1 – Les 4 types de relations entre les classes

Présentation des relations

- Les objets, et les classes qui leur correspondent, peuvent avoir des relations entre elles.
- Il y a principalement 4 types de relations entre les classes :

  - L'agrégation
  - La composition
  - L'énumération
  - L'héritage

- Agrégation et composition sont 2 relations assez semblables : c'est une relation « avoir ».
- L'énumération est une sorte de composition.
- L'héritage est une relation à part : c'est une relation « est un ».
- On va expliquer ce que sont ces 4 relations avec des exemples. Il faut déjà retenir le vocabulaire.

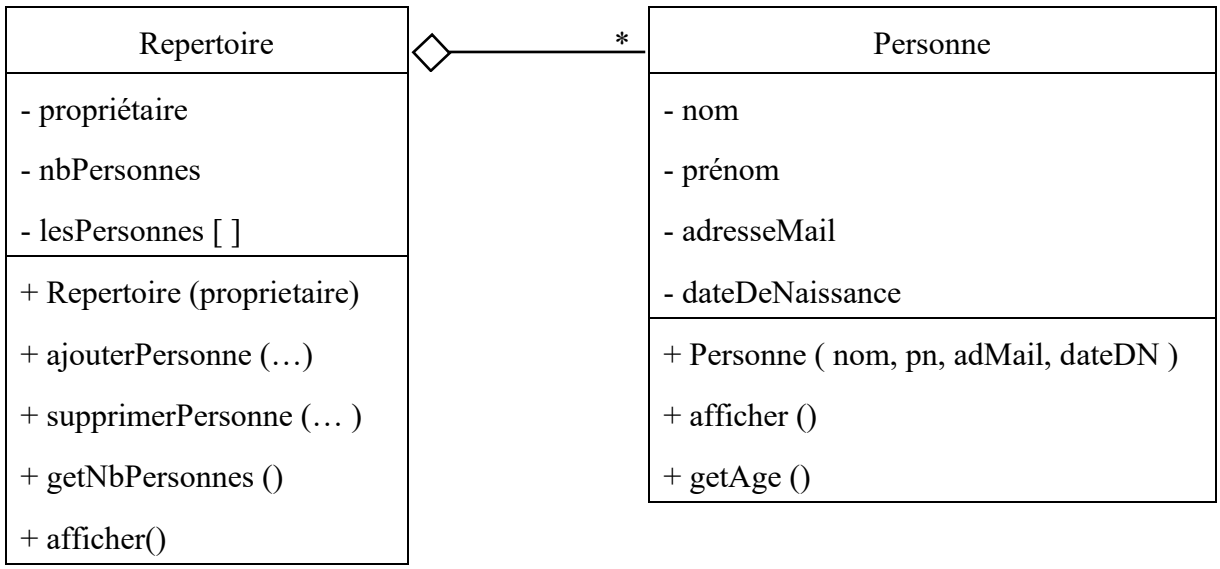
2 - Agrégation

Rappel : il y a principalement 4 types de relations entre les classes :

- L'agrégation
- La composition
- L'énumération
- L'héritage

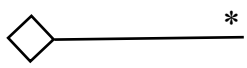
Exemple d'agrégation : un répertoire de personnes

Diagramme de classes UML



- Le répertoire **agrège** des personnes. Cela veut dire qu'il **contient une liste** (collection, tableau, etc.) de personnes.
- C'est une **agrégation** et pas une composition. Cela veut dire que **si on supprime le répertoire**, on ne doit **pas forcément supprimer les personnes** : elles sont peut-être utilisées dans un autre répertoire.
- Syntaxe UML de l'agrégation :

  ⇒ La relation entre Repertoire et Personne matérialisée graphiquement par le lien :

  ⇒ on met un losange du côté de **l'agrégat** (le **conteneur)** : ◇

  ⇒ on met une étoile du côté des **éléments agrégé (**les **contenus)** : « * ». « * » veut dire plusieurs : le répertoire agrège (contient) plusieurs personnes.

- Subtilités de syntaxe :

  ⇒ La collection s'appelle: « lesPersonnes ». C'est une convention simple : « les » pour la collections, suivi du nom de la classe. Des crochets « [ ] » pour montrer que c'est une collection.

  ⇒ On n'est pas obligé de la mettre dans le diagramme de classe : la relation de composition dit qu'elle existe.

  ⇒ On met les paramètres des méthodes si c'est facile, rien s'il n'y en a pas.

  ⇒ On met « … » comme paramètres de méthodes si on ne sait pas encore ce qu'ils seront.

- Relation AVOIR

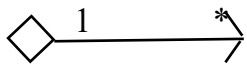
  ⇒ Une agrégation, comme une composition, ou les relations entre classes qui ne sont pas des héritages, est une relation AVOIR :

  ⇒ Un répertoire « a des » personnes.

  ⇒ Une personne a 1 répertoire (ou plusieurs, comme on veut).

- Subtilités UML

  ⇒ On ne s'intéresse pas aux répertoires des personnes, mais aux personnes dans le répertoire : la relation ne va que dans un sens.



  ⇒ On peut considérer que l'agrégation c'est le 1 et la flèche par défaut.

Présentation de l'exemple

- On veut manipuler dans notre programme un répertoire de personnes.
- Le **répertoire** a un **propriétaire** identifié par son nom. Le propriétaire est une information fixée à la création et non modifiable. Il contient des personnes.

> - On peut **ajouter** et **supprimer** des personnes dans le répertoire.
> - On peut aussi **modifier** les informations d'une personne.
> - On peut récupérer le **nombre de personnes**.
> - On peut **afficher** la liste de toutes les personnes du répertoire avec le nom du propriétaire et le nombre de personnes. On affiche une ligne par personne avec toutes les informations de la personne et son âge plutôt que sa date de naissance.

- Une personne a un nom, un prénom, une adresse mail et une date de naissance.

> - On peut **afficher** les informations de chaque personne (toutes les informations sur une seule ligne, avec son **âge** plutôt que sa date de naissance).

- On va **écrire un programme** qui permet de :

> - Créer et afficher un répertoire dont vous êtes le propriétaire.
> - Mettre 3 personnes dedans et afficher le répertoire.
> - Supprimer la deuxième personne créée puis afficher le répertoire.

Exemple d'affichage des résultats attendus :

```
Affichage du répertoire après la création du répertoire :
propriétaire : Bertrand : 0 Personnes

Ajout de 3 personnes et affichage du répertoire :
propriétaire : Bertrand : 3 Personnes
toto - lolo - toto@gmail.com - 19
titi - lili - titi@gmail.com - 19
tutu - lulu - tutu@gmail.com - 18

Affichage du répertoire après suppression :
propriétaire : Bertrand : 2 Personnes
toto - lolo - toto@gmail.com - 19
tutu - lulu - tutu@gmail.com - 18
```

Codage de l'agrégation

La classe Personne : Rien à signaler, ni en pseudo-code, ni en Java, ni en python

La classe Repertoire : pseudo-code

- ***Surcharge de la méthode ajouter()***

On va avoir **2 méthodes ajouterPersonne() avec des paramètres différents**. C'est ce qu'on appelle une « **surcharge** ». C'est très pratique.

Première version : on passe les informations d'une personne et on instancie la personne dans la méthode :

```
ajouterPersonne( nom, prenom, adMail, annee ) :
    lesPersonnes.add( new Personne(nom, prenom, adMail, annee) )
```

Deuxième version : on passe une personne en paramètre :

```
ajouterPersonne( Personne p ) :
    lesPersonnes.add(p)
```

A noter qu'on se dote d'une méthode « add » qui permet d'ajouter un élément dans la collection.

- ***méthode supprimerPersonne(Personne p)***

Pour la méthode supprimer(), on passe en paramètre l'objet personne qu'on veut supprimer.

```
supprimerPersonne(Personne p) :
    lesPersonnes.remove(p)
```

A noter qu'on se dote d'une méthode « remove» qui permet de supprimer un élément dans la collection.

La classe Repertoire : code Java

On a une classe Java pour le type de l'attribut lesPersonnes : classe ArrayList. Notez la syntaxe avec < > : private ArrayList<Personne> **lesPersonnes**;

On initialise l'attribut lesPersonnes à l'instanciation avec un new. Notez la syntaxe.

On a deux méthodes ajouterPersonnes() avec des paramètres différents.

Les méthode add et remove sont du Java.

```
import java.util.ArrayList;
public class Repertoire {
   private String proprietaire;
   private ArrayList<Personne>   lesPersonnes;

   public Repertoire(String proprietaire) {
      this.proprietaire = proprietaire;
      lesPersonnes = new ArrayList<Personne>();
   }
   public void ajouterPersonne(String nom, String prenom, String adMail, int
     annee) {
      lesPersonnes.add(new Personne(nom, prenom, adMail, annee));
   }
   public void ajouterPersonne(Personne p) {
      lesPersonnes.add(p);
   }
…
}
```

La classe Repertoire : code Python

Il n'y a **pas de surcharge basique** en python.

Dans un premier temps, on ajoute une méthode « **ajouterPersonne2()** » : ce n'est pas pratique. Il y a des solutions techniques plus complexes pour améliorer cette situation.

Le reste est « basique ». On utilise **les [ ] pour avoir la collection** de base du Python qui donne accès aux méthodes **append** (pour add) et remove.

Le fichier doit commencer par importer le fichier de la classe Personne

```
from Personne import *       # on importe le fichier de la classe Personne
class Repertoire:
   def __init__(self, proprietaire):
      self.proprietaire = proprietaire
      self.lesPersonnes = []

def ajouterPersonne(self, nom, prenom, adMail, anneeNaissance):
     self.lesPersonnes.append(Personne(nom, prenom, adMail, anneeNaissance))

def ajouterPersonne2(self, personne): # pas de surcharge simple en Python
     self.lesPersonnes.append(personne)
…
```

3 - Notion de tests unitaires

Présentation

- Avec l'agrégation on voit un premier **exemple de code avec 2 classes**. Le programme va utiliser ces deux classes et leurs méthodes.
- Comment vérifier que chaque classe fonctionne correctement ?
- C'est la notion de **tests unitaires**.
- Pour chaque classe on va créer une classe en plus qui sera une classe de « tests unitaires »
- On appelle la classe de tests unitaires « **ClasseTU** ». Cette classe contient un « main » et s'occupe de tester toutes les méthodes de la classe pour vérifier que tout fonctionne.

Exemple

- une classe « **PersonneTU** » pour les tests unitaires de la Personne.
- une classe « **RepertoireTU** » pour les tests unitaires du Repertoire. RepertoireTU correspond finalement au programme demandé qui est très simple et ne fait que des petits tests.

Tests unitaires de PersonneTU

- ***pseudo-code de PersonneTU***

Il faut tester **le constructeur**, **afficher**() et **getAge**(). On peut donc simplement écrire :

```
# Tests unitaires de Personne
personne=Personne("toto", "lolo", "toto@gmail.com", 2000);
personne.afficher();
print("age : "+personne.getAge())
```

- ***Java***

```
public class PersonneTU{
   public static void main (String args[]){
      Personne personne=new Personne("toto", "lolo", "toto@gmail.com", 2000);
      personne.afficher();
      System.out.println(("age : "+personne.getAge());
   }
}
```

- ***Python***

```
from Personne import *      # on importe le fichier de la classe Personne
personne=Personne("toto", "lolo", "toto@gmail.com", 2000);
personne.afficher();
print("age : "+personne.getAge())
```

- ***Conclusion***

C'est presque pareil en pseudo-code, Java ou Python. Il n'y a que la structure qui change.

Tests unitaires de RepertoireTU

- ***pseudo-code de RepertoireTU***

Il faut tester **le constructeur**, et **toutes les méthodes**. Cela correspond au programme demandé.

```
# Tests unitaires de ReperoireTU
print("Affichage du répertoire après l'instanciation : ")
rep=new Repertoire("Bertrand")
rep.afficher()

print("Ajout de 3 personnes et affichage du répertoire : ")
rep.ajouterPersonne("toto", "lolo", "toto@gmail.com", 2000)
p = new Personne("titi", "lili", "titi@gmail.com", 2000)
rep.ajouterPersonne(p)
rep.ajouterPersonne(new Personne("tutu","lulu","tutu@gmail.com",2001) )
rep.afficher()

print("Affichage du répertoire après suppression : ")
rep.supprimerPersonne(p)
rep.afficher()
```

- ***code Java et code Python***

Il n'y a rien de particulier en plus dans ces codes par rapport au pseudo-code.

4 - Notion de surcharge

Principes

- En POO, on peut définir des méthodes dans une même classe avec le même nom mais des paramètres différents : c'est ce qu'on appelle **la surcharge.**

  ⇒ La surcharge peut s'appliquer au constructeur.

- C'est très pratique et rend l'écriture et l'usage des codes simplifiés.

Exemples

Le constructeur de la classe propriétaire

- Le premier constructeur envisagé crée un répertoire vide :

```
Repertoire (proprietaire)
```

- On pourrait se doter d'un 2ème constructeur permettant de créer un répertoire en le remplissant avec une liste de personnes qui existe déjà :

```
Repertoire (proprietaire, lesPersonnes[] )
```

La méthode « ajouterPersonne() »

- Une version avec un objet personne passée en paramètre :

```
ajouterPersonne(personne)
```

- Une version avec les paramètres d'une personne passés en paramètres de la méthode :

```
ajouterPersonne(nom, prenom, adresseMail, anneeNaissance)
```

Code Java et code Python

Code Java

- L'écriture se fait « tout naturellement » : il suffit d'écrire un deuxième constructeur ou une deuxième méthode avec les nouveaux paramètres.

Code Python

- Le python de permet pas de surcharge à proprement parler (2 méthodes avec le même nom et des paramètres différents). Mais il permet d'avoir une méthode avec une liste variable de paramètres :

```
def ajouterPersonne(self, nom=None, prenom=None, adMail=None,
                                    anneeNaissance=None,
        personne=None):
    # usage 1 : ajouterPersonne(nom, prenom, adMail, anneeNaissance)
    # usage 2 : ajouterPersonne(personne=personne)
    if personne:
        self.lesPersonnes.append(personne)
    else:
        self.lesPersonnes.append(Personne(nom, prenom, adMail, anneeNaissance))
```

  - ***usage :***

```
repertoire.ajouterPersonne("toto", "lolo", "toto@gmail.com", 2000);
p = Personne("titi", "lili", "titi@gmail.com", 2000);
repertoire.ajouterPersonne(personne=p);
```

- Notez le (personne = p)
- Le Python offre d'autres façon de gérer des listes de paramètres variables (kwargs).

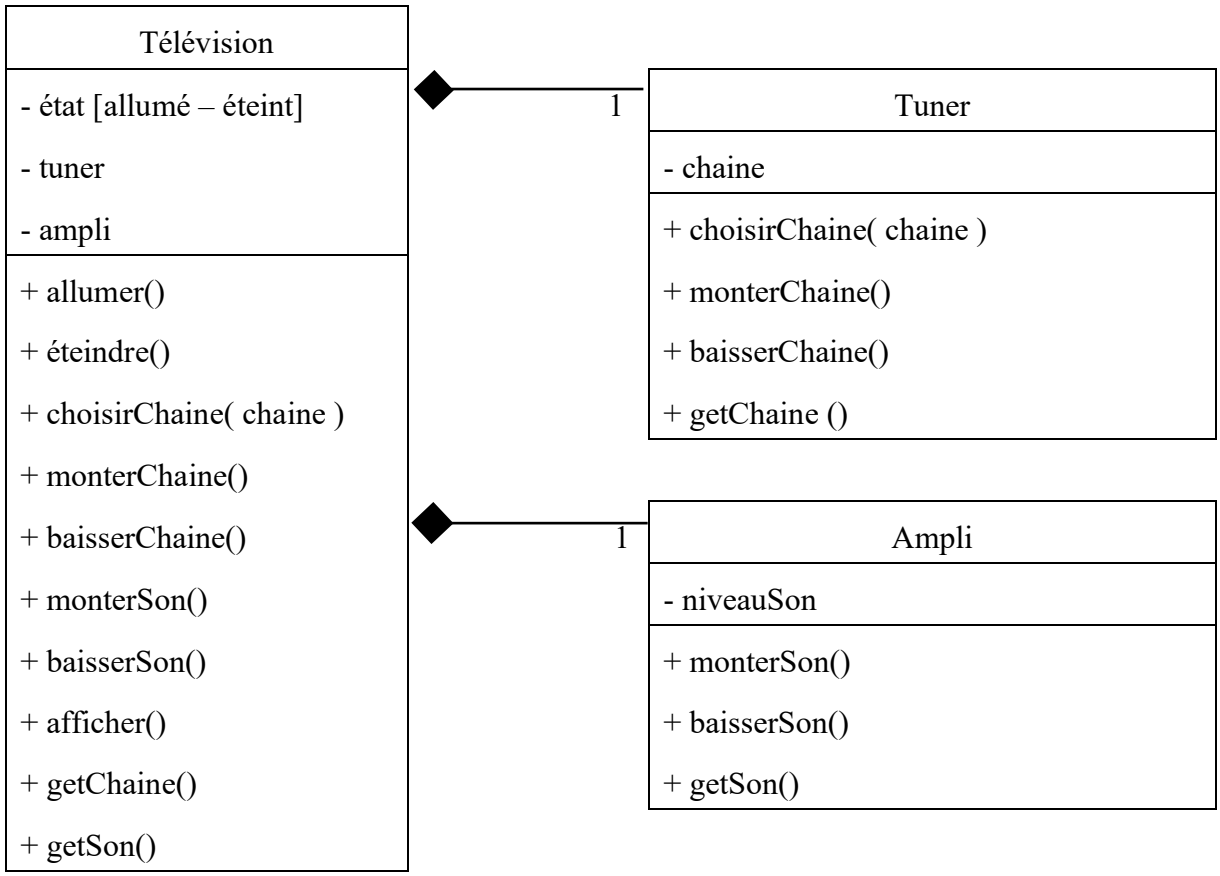
5 - Composition

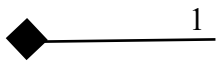
Rappel : il y a principalement 4 types de relations entre les classes :

- L'agrégation
- La composition
- L'énumération
- L'héritage

Exemple de composition : la télévision et ses composants (tuner et ampli)

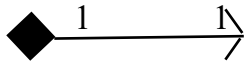
Diagramme de classes



- La classe Télévision est composée d'un objet « tuner » et d'un objet « ampli ».
- Cette relation est matérialisée dans le diagramme par le lien : 

Principes de la composition

- La télévision est composée d'un ampli et d'un tuner. Cela veut dire qu'elle contient un ampli et un tuner.
- C'est une **composition** et pas une agrégation. Cela veut dire que **si on supprime la télévision**, **on supprime aussi l'ampli et le tuner**. L'ampli et le tuner ne sont utilisés que par la télévision.
- Syntaxe UML :
  - ⇒ on met un losange du côté du **composé** (le **conteneur)** : ◆
  - ⇒ on met une 1 du côté des **composants**, les **contenus** : la télévision contient 1 tuner et 1 ampli. Ca pourrait être 2, 3, plusieurs.
- Subtilités
  - ⇒ On ne met **rien du côté du composé** : c'est toujours 1 par défaut.
  - ⇒ Si on ne met **rien du côté du composant**, c'est 1 par défaut.
- Relation AVOIR
  - ⇒ Une composition, comme une agrégation, ou les relations entre classes qui ne sont pas des héritages, est une relation AVOIR :
  - ⇒ Une télévision « a un » tuner.
  - ⇒ Un tuner « a 1 » télévision.
- Subtilités UML
  - ⇒ On ne s'intéresse pas à la télévision du tuner, mais au tuner de la télévision : la relation ne va que dans un sens.



  - ⇒ On peut considérer que la composition, c'est les 1 et la flèche par défaut.

Présentation de l'exemple

- On veut **manipuler une télévision** (exemple déjà abordé en exercice).
- On peut allumer et éteindre la télé, monter et baisser le son, changer de chaine en donnant le numéro de la chaine, monter ou baisser le numéro de la chaine.
- La télévision est composée d'un tuner qui gère les chaines et d'un ampli qui gère le son.
- On veut gérer séparément les objet « tuner » et « ampli ».
- On veut écrire un programme qui permet de tester la télévision :

> - Afficher l'état général après l'instanciation.
> - Allumer et afficher l'état général.
> - Choisir une chaine (la 5), monter le son 3 fois et afficher l'état général.
> - Monter la chaine 3 fois, baisser la chaine 1 fois, baisser le son et afficher l'état général.
> - Eteindre et afficher l'état général.

Résultats attendus

```
off : chaine =  1 - son : 3
on : chaine =  1 - son : 3
on : chaine =  5 - son : 6
on : chaine =  7 - son : 5
off : chaine =  7 - son : 5
```

Codage de la composition

Les classes Ampli et Tuner : Rien à signaler, ni en pseudo-code, ni en Java, ni en python

La classe Television : pseudo-code

- ***Constructeur : on instancie les composants***

Le constructeur instancie ses composants :

```
Television() :
    etat = eteint
    ampli = new Ampli()
    tuner = new Tuner()
```

- ***Délégation***

Les méthodes correspondant aux composants « délèguent » le travail aux composants :

```
monterSon() :
    ampli.monterSon()

choisirChaine( chaine )
    tuner.choisirChaine (chaine)

getSon( ):
     return ampli.getSon()
etc.
```

Code Java et code Python

- Rien de particulier à signaler. Regarder directement l'exemple.

Code des tests unitaires

3 classes à tester

- Ici, on peut faire des tests unitaires sur les 3 classes : Télévision, Ampli et Tuner.
- On peut donc avoir 3 classes de tests unitaires : TelevisionTU, AmpliTU, TunerTU, avec leurs « main » pour faire les tests.
- La classes TelevisionTU correspond au programme de tests qu'on veut écrire dans l'exemple.

La classe TelevisionTU : pseudo-code

Le pseudo code est très simple et très simple à traduire en Java ou en Python

```
television = Television();
television.afficher();

television.allumer();
television.afficher();

television.choisirChaine(5);
television.monterSon();
etc.
```

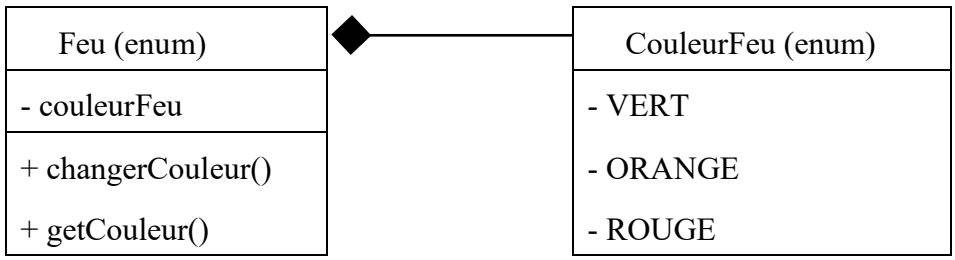
6 - Enumération

Rappel : il y a principalement 4 types de relations entre les classes :

- L'agrégation
- La composition
- L'énumération
- L'héritage

Classe énumération

Exemple



Technique

- On définit l'énumération comme une classe particulière : une classe d'énumération.
- Les attributs de l'énumération sont les valeurs possibles.
- Il n'y a pas de méthodes dans l'énumération.

Principes

- En POO, on utilise souvent des classes d'énumération qui nous permettent de définir des petites listes de valeurs constantes.
- Ces valeurs sont utilisées pour fixer les valeurs possibles pour un attribut et ainsi éviter les erreurs.
- L'énumération devient alors un type pour un attribut.

Code Python

```
from enum import Enum
class CouleurFeu(Enum):
    VERT = 0;
    ORANGE = 1;
    ROUGE = 2;
```

```
class FeuTricolore:
    def __init__(self):
        self.couleur = CouleurFeu.ROUGE;
    def changerCouleur(self):
        if self.couleur == CouleurFeu.VERT :
            self.couleur = CouleurFeu.ORANGE;
        elif …
```

```
# main
feu = FeuTricolore();
print("Couleur : ", feu.getCouleur());
if(feu.getCouleur()==CouleurFeu.ROUGE):
    print("Le feu est rouge, il faut s'arrêter")
```

Code Java

```
public enum CouleurFeu{
  VERT,
  ORANGE,
  ROUGE
}
```

```
public FeuTricolore(){
  this.couleur = CouleurFeu.ROUGE;
}

public void changerCouleur(){
  if (this.couleur == CouleurFeu.VERT){
    this.couleur = CouleurFeu.ORANGE;
  }else if …
```

```
# main
feu = new FeuTricolore();
System.out.println("Couleur : ", feu.getCouleur());
if(feu.getCouleur()==CouleurFeu.ROUGE)
      System.out.println ("Le feu est rouge, il faut s'arrêter")
```

UML - Exercices

Principes

- Pour tous les exercices, il d'agit de créer des classes avec un constructeur et d'écrire le main en utilisant ces classes. On précise bien les paramètres des méthodes.
- On fait le diagramme de classes avec le constructeur et les paramètres des méthodes.

Agrégation et composition : la voiture

- On reprend un exercice précédent.

  - Une voiture a une marque, un modèle, une immatriculation. Elle peut avoir 2 portes, 4 portes ou 5 portes.
  - On peut mettre le moteur de la voiture en route et passer des vitesses. Le passage de vitesse se fait forcément de 1 en 1, en montant ou en descendant. On peut aussi passer au point mort à tout moment. On peut aussi freiner ou accélérer.

- On ajoute les éléments suivants :

  - La voiture a un moteur qui a un nom et une puissance.
  - La voiture a des roues (2 roues avant, 2 à l'arrière et 1 de secours). Chaque roue a un type de pneu, une marque de pneu, et le pneu peut être neuf ou usagé ou crevé. La position des roues sera gérée avec une énumération.
  - On veut pouvoir changer une roue en cas de crevaison.
  - On veut pouvoir récupérer l'état de sa roue de secours.
  - La voiture peut « crever ».

- On testera la voiture ainsi :

  - On crée une voiture (une Renault Espace 5 portes moteur v6 245 chevaux avec des pneus Dunlop de type mixte, tous neufs). On affiche ses caractéristiques.
  - On démarre. On passe les vitesses jusqu'en 3ème. On affiche la vitesse en cours à chaque changement de vitesse.
  - On crève la roue avant droite. On freine et on passe au point mort. On affiche les caractéristiques de la voiture.
  - On change la roue crevée. On affiche les caractéristiques de la voiture.

Agrégation et composition : Personnage de jeu vidéo

- On reprend un exercice précédent.

> - Un personnage de jeu vidéo a un nom, une force, une localisation, une expérience et des dégâts.
> - Le personnage peut se déplacer et combattre un autre personnage.

- On y ajoute le fait qu'un personnage peut trouver des armes et les accumuler dans un stock. Une arme a un nom, une puissance et une valeur.
- On écrit un programme qui permet de :

> - Créer un joueur et afficher ses caractéristiques avec l'arme qu'il porte (ou pas).
> - Créer 3 armes (on considère qu'elles sont sur le terrain et que le joueur les trouve).
> - Le joueur ajoute ces 3 armes à son stock. Affichez le stock.
> - Le joueur sélectionne la dernière arme dans son stock et porte cette arme (quand il porte une arme, elle sort de son stock). Affichez les caractéristiques du joueur avec l'arme qu'il porte et affichez le stock de ses armes (l'arme qu'il porte n'est plus dans le stock).
> - Le joueur veut changer d'arme. Il sélectionne à nouveau la dernière arme dans son stock (ce n'est pas la même que précédemment) et remplace l'arme qu'il porte par ce nouveau choix.
> - Le joueur combat puis il range son arme. Affichez les caractéristiques du joueur et son stock d'armes.
> - Le joueur se sépare de la première arme de son stock. Affichez le stock.

Agrégation et composition : le Distributeur

- On reprend un exercice précédent.
- On va gérer une classe boisson séparément du distributeur. On gère 2 boissons : un café à 40 centimes et un chocolat à 50 centimes.
- Le but est d'obtenir les résultats suivants :

```
Le distributeur encaisse 20 centimes.
niveau de sucre baissé : 1
niveau de sucre baissé : 0
Le distributeur prépare la boisson n°1
Le montant encaissé est trop faible
Le distributeur encaisse 50 centimes.
Le distributeur prépare la boisson n°1
café sucre=0 en cours de préparation
Prix:40 - payé:70 - rendu:30

Le distributeur encaisse 10 centimes.
Le distributeur encaisse 20 centimes.
Le distributeur rend 30 centimes.

Le distributeur encaisse 50 centimes.
Le distributeur prépare la boisson n°3
La boisson 3 n'est pas disponible.
Le distributeur prépare la boisson n°2
chocolat sucre=2 en cours de préparation
Prix:50 - payé:50 - rendu:0
```

2 - HERITAGE – POLYMORPHISME – STATIC – CLASSE ABSTRAITE - INTERFACE

1- Héritage

Rappel : il y a principalement 4 types de relations entre les classes :

- L'agrégation
- La composition
- L'énumération
- L'héritage

Exemple conduisant à l'héritage

- On a une classe Chat et une classe Chien :

Chat
- nom - age - propriétaire - raceChat
+ crier() + manger() + ronronner()

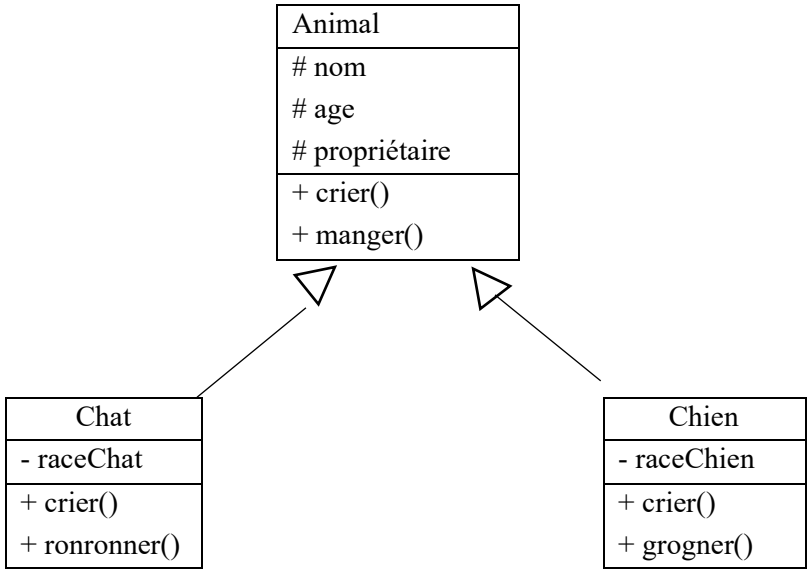
Chien
- nom - age - propriétaire - raceChien
+ crier() + manger() + grogner()

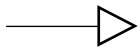
- On voit que les 2 classes ont des attributs et des méthodes en commun.
- Concernant les méthodes, on peut considérer que **manger()** est la même méthode pour le chat et pour le chien.
- Par contre, **crier()** est différente : le chat miaule et le chien aboie !

Principe de l'héritage : mettre le commun dans une classe de base

- Dès que 2 classes ont des attributs ou des méthodes en commun :
  - ⇒ on retire ces attributs et méthodes de leurs classes de départ pour créer une nouvelle classe avec les attributs et méthodes communes.
  - ⇒ On fait ensuite « hériter » les classes de départ de la nouvelle classe.

- Les classes Chat et Chien héritent de la classe Animal :



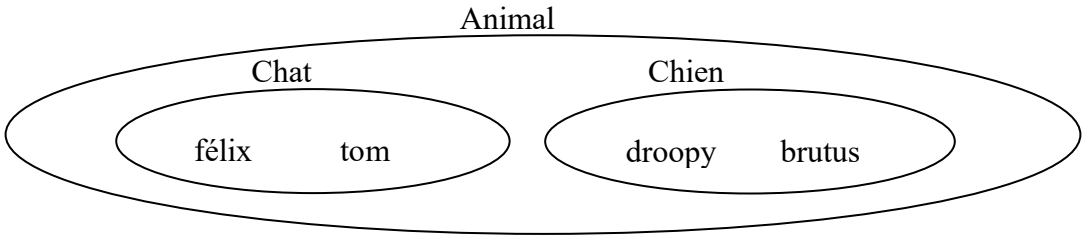
- La nouvelle classe est appelée : **classe de base** (ou **classe-mère** ou **classe-parent**).
- Les classes de départ sont appelées : **classe dérivée** (ou **classe-fille** ou **classe-enfant**).
- On dit que **les classes-enfant « héritent » de la classe parent.** En effet, tous les attributs et les méthodes de la classe-parent seront accessibles par la classe enfant.
- La flèche avec un triangle blanc au bout est une flèche d'héritage : 

Inclusion ensembliste, relation « est-un »

Inclusion ensembliste

L'inclusion ensembliste est ce qui traduit le mieux la notion d'héritage.

Un ensemble correspond à une classe. Un élément d'ensemble correspond à un objet. Un ensemble inclut dans un autre correspond à un héritage.



Relation « est un »

L'héritage, la flèche à triangle, l'inclusion sont des **relations « est un »**.

Un Chat « est un » animal. Félix « est un » Chat. félix est un Animal. Etc.

Utilité

L'héritage est utilisé pour :

- **Factoriser le code** : ça évite d'avoir à écrire deux fois la même chose !
- **Utiliser un code existant déjà** : c'est une forme de factorisation.
- **Le polymorphisme** (possibilité qu'une même méthode se réalise différemment).

Mise en œuvre technique

Encapsuation : 1 nouveau niveau de visibilité : protected

Symbole	Mot-clé	Signification
-	private	Visible uniquement **dans la classe**
#	protected	Visible dans la classe et **dans les classes dérivées**
+	public	Visible **partout**

Redéfinition de méthode

- Dans la classe dérivée, **on peut redéfinir une méthode déjà présente** dans la classe de base**.**
- **Il vaut mieux n'hériter que d'une classe** pour éviter les incohérences. Une classe a un seul parent direct et pas plusieurs. **On évite donc l'héritage multiple** (interdit en Java, possible en Python).

Constructeur

- Constructeur de Animal : **Animal** (nom, age, propriétaire)
- Constructeur de Chat : **Chat** (nom, age, propriétaire, raceChat)
- Constructeur de Chien : **Chien** (nom, age, propriétaire, raceChien)

Le constructeur de l'enfant (le chat, le chien) reprend les attributs du parent (l'animal).

Principes du codage de l'héritage

Dire le lien d'héritage : faire le lien entre la classe enfant et la classe mère

- Java :

```
public class Chat extends Animal {
}
```

- Python :

```
class Chat(Animal) :
```

Encapsulation

- En Java, les attributs sont souvent « protected ».
- En Python, il n'y a pas d'encapsulation.

Accéder à une méthode du parent : mot clé « super »

- On utilise le **mot clé « super »** pour accéder à une méthode ou au constructeur de la classe parent.
- Pour utiliser la méthode afficher() de la classe Animal dans une méthode de la classe Chat, on écrit : **super.afficher()**
- Il y a des variantes entre le Java et le Python.

Code des classes

La classe Animal : rien à signaler, ni en pseudo-code, ni en Java, ni en python

Les classes Chat et Chien : pseudo-code

- ***Constructeur : on instancie le parent***

```
Chien (self, nom, age, proprietaire, raceChien):
    super(nom, age, proprietaire)  // on instancie le parent, l'Animal
    this.raceChien = raceChien
```

- ***Appel à une méthode du parent : afficher()***

```
afficher() : // afficher du Chien
    super.afficher()          // on appelle la méthode afficher du parent, l'Animal
    print("Je suis un chien de race : "+this.raceChien)
```

Code Java

- Rien de particulier par rapport au principe général. Regarder le code complet.

Code Python

- En Python, l'instanciation du parent se fait par sa méthode __init__
- En Python, le mot clé super est écrit avec des parenthèses.

➢ Exemple de la classe Chat :

```
class Chat(Animal):
    def __init__(self, nom, age, proprietaire, raceChat):
        super().__init__(nom, age, proprietaire)
        self.raceChat = raceChat
    def afficher(self):
        super().afficher()
        print("Je suis un chat de race : "+self.raceChat)
    def crier(self):
        print("miaou")
    def ronronner(self):
        print("ronron")
```

Tests Unitaires

- Pour tester les classes, on va instancier un objet de chaque classe et faire quelques appels de méthodes.
- On applique les méthodes possibles selon la classe instanciée.

```
animal=Animal("Marsupilami", 67, "André Franquin")
animal.afficher()

felix=Chat("Felix",100, "Otto Mesmer", "de gouttière")
felix.afficher()
felix.crier();
felix.ronronner();

droopy=Chien("Droopy", 82, "Tex Avery", "Basset Hound")
droopy.afficher()
droopy.crier();
droopy.grogner();
```

2 - Polymorphisme

Principe de substitution

- Dans le code, si on remplace un objet d'une classe parent par un objet d'une classe enfant, ça va continuer à marcher (compilation et exécution).
- En effet, toutes les méthodes accessibles par un objet d'une classe parent seront accessibles par un objet d'une classe enfant, et donc le code continuera à marcher.
- Cette possibilité de substitution (remplacement), **c'est ce qu'on appelle le principe de substitution**.

Principe de substitution :

On peut substituer :

un **objet d'une classe mère** (ou classe de base)

par

un **objet d'une classe enfant** (ou classe dérivée ou sous-classe).

C'est ce qui permettra le polymorphisme.

Polymorphisme

Principes

- Il y a polymorphisme quand la méthode appelée est différente selon l'objet effectivement présent.
- Le fait que la méthode soit différente fait que le comportement du programme change bien que le code écrit reste identique.
- Ce changement de comportement, c'est ce qu'on appelle du polymorphisme ou un comportement polymorphe
- C'est rendu possible par le principe de substitution.

Par exemple :

- si on écrit : **animal.afficher( )**, on pense que c'est la **méthode afficher() de la classe animal** qui va s'exécuter.
- Mais si animal est en réalité un chat, ce sera la méthode afficher() de la classe Chat qui va s'exécuter.
- Et si animal est en réalité un chien, ce sera la méthode afficher() de la classe Chien.
- C'est ce qu'on appelle du polymorphisme ou un comportement polymorphe.

Code polymorphe

```
// on a trois objets : marsupilami qui est un animal, felix un chat, droopy un chien
tabAnimaux=[]
tabAnimaux.append(marsupilami)
tabAnimaux.append(felix)
tabAnimaux.append(droopy)
pour i de 1 à taille(tabAnimaux)
   print("----------------------")
   tabAnimaux[i].afficher()     // instruction polymorphe
   tabAnimaux[i].crier();       // instruction polymorphe
```

Résultats d'un code polymorphe

```
nom: Marsupilami - age: 67 - proprietaire: André Franquin
je crie !!!
----------
nom: Felix - age: 100 - proprietaire: Otto Mesmer
Je suis un chat de race : de gouttière
miaou
----------
nom: Droopy - age: 82 - proprietaire: Tex Avery
Je suis un chien de race : Basset Hound
ouah
```

3 - Objet et référence

Identité de l'objet – référence de l'objet

- Chaque objet possède une **identité** attribuée de manière implicite à la création de l'objet et qui n'est jamais modifiée (c'est son **adresse en mémoire**).
- De ce fait, chaque objet a forcément une identité distincte d'un autre objet et est donc forcément différent d'un autre objet.
- On accède à l'identité de l'objet par une variable avec un nom qu'on appelle aussi référence.
- **Au sens strict, l'objet, c'est l'identité.** La référence c'est une variable qui amène à l'objet.

REFRENCE		IDENTITE
nom_1	⟶	un objet en tant qu'identité
nom_2	⟶	un autre objet en tant qu'identité

- Techniquement, l'objet c'est la mémoire qui permet de stocker l'état et les comportements de l'objet. L'identité de l'objet c'est cette mémoire.
- Notons que deux objets différents ayant une identité différente peuvent avoir le même état.

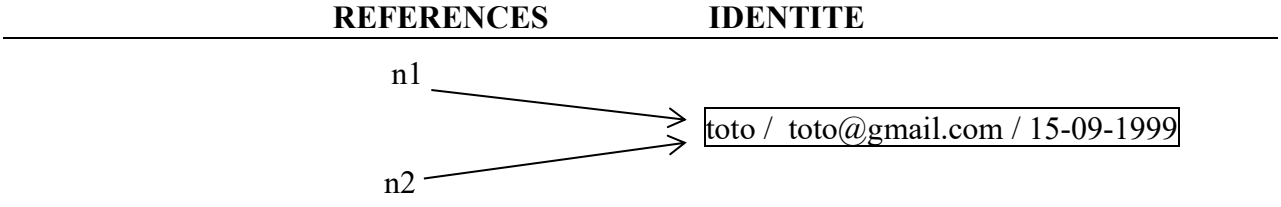
Une identité unique – plusieurs références possibles

Références possibles d'un objet

- Un objet peut être référencé par une variable.
- Un objet peut être référencé par un élément d'un tableau.
- Un objet peut être référencé par un attribut d'un autre objet

Références multiples d'un même objet

- On peut avoir plusieurs références pour un même objet.
- Il peut y avoir plusieurs variables qui font référence au même objet.
- On peux avoir deux références n1 et n2 pour un même objet (une même identité) :



➢ Exemple de code aboutissant à ça :

```
n1 = new Personne(« toto », « toto@gmail.com », « 15-09-1999 »)
n2=n1
```

Ici, on n'a pas dupliqué l'objet en faisant n2=n1. On s'est juste doté d'une 2ème référence vers l'objet en tant qu'identité.

Pour dupliquer l'objet, il faut faire une nouvelle instanciation.

4 - Attributs et méthodes de classes

Instance vs Classe

- Les attributs et les méthodes vues jusqu'à présent sont des **attributs et des méthodes d'instance** : ils s'appliquent à chaque objet instancié d'une classe.
- Les attributs et les méthodes de classes sont des attributs et des méthodes qui sont définis au niveau d'une classe et qui n'ont pas besoin d'un objet pour être utilisés.

Les deux usages

- On peut définir 2 usages des attributs et des méthodes de classes :

1 : propriété concernant toute la classe

- Attributs et méthodes de classe permettent de **définir des caractéristiques générales à la classe** qui sert normalement à instancier des objets.
- Par exemple, on peut définir un attribut de classe qui donnerait le nombre d'objets instanciés pour une classe et sa méthode de classe associée qui « get » cet attribut.

2 : bibliothèque de méthodes dans une classe sans objet

- Attributs et méthodes de classes permettent de définir des fonctions qui seront utilisées indépendamment de tout objet de la classe.
- Dans ce cas, la classe ne sert pas à instancier des objets mais uniquement à regrouper des méthodes utilitaires et des attributs qui seront des constantes : c'est une **bibliothèque de fonctions** sur le même thème qui sont regroupées dans une classe.

Exemple usage 1 en Java : mot clé « static »

```
public class Gaufre {
    // attributs et méthodes de classe
    private static int nbGaufres=0;
    public static int getNbGaufres(){
        return nbGaufres;
    }
    // attributs et méthodes d'objet (ou d'instance)
    private String pate;
    private String garniture;
    ...
    public static void main (String args[]) {
        Gaufre gaufre1=new Gaufre("standard");
        gaufre1.afficher();
        System.out.println(Gaufre.getNbGaufres()+" gaufre(s) instanciées");

        Gaufre gaufre2=new Gaufre("bio");
        gaufre2.afficher();
        System.out.println(Gaufre.getNbGaufres()+" gaufre(s) instanciées");
    }
}
```

Exemple usage 2 en Python : décorateurs @classmethod et @staticmethod

```
import sys
class Comparaisons :
  seuil = 10;                  # attribut de classe
  @classmethod                 # methode de classe, technique 1
  def min(cls, a, b) :         # cls (classe) plutot que self
    print("min2:"+str(cls))  # pour voir le contenu de cls
    if (a < b):
      return a;
    else:
      return b;
  @staticmethod                # methode de classe, technique 2, pas de cls
  def grand(a) :               # pas de cls dans les paramètres
     return (int(a) > Comparaisons.seuil);
# main
n = int(sys.argv[1]);
p = int(sys.argv[2]);
print("Le plus petit des 2 premiers : " + str(Comparaisons.min(n, p)));
if (Comparaisons.grand(n)):
  print(str(n) + " est grand que le seuil qui vaut " + str(Comparaisons.seuil));
```

5 - Classe abstraite - Interface

Classe abstraite

Définition

- **Une classe abstraite** est une classe qui **ne peut pas être instanciée** : on ne peut pas créer d'objet correspondant à cette classe. Ce n'est donc pas un moule !

Pourquoi faire ?

- La classe Animal est utile pour **factoriser ce qui est commun**.
- La classe Animal pourrait être rendue abstraite pour éviter qu'on puisse créer des objets à partir de cette classe.
- La classe abstraite devient une partie du moule de la classe enfant.
- Ca permet de factoriser du code et d'être plus proche du réel (il y a des animaux, des chats et des chiens).

Dans le code

- En Java, on ajoute le mot clé « abstract » devant le mot clé « class ».
- En Python… le Python ne gère pas les classes abstraites. Il faudra faire « comme si ».

Méthodes abstraites

Définition

- Une méthode abstraite est une méthode qui n'a qu'une en-tête mais pas de corps.
- Quand une classe contient une méthode abstraite, elle est forcément abstraite.

A quoi sert-elle ?

- Elle sert à obliger les classes enfants qui hériteront de la classe mère abstraite à définir la méthode abstraite. C'est donc une obligation qui est faite au programmeur qui utilise la classe abstraite.

Pourquoi faire ?

- Si on reprend le **modèle** Animal – Chat – Chien, on a la méthode crier () qui se trouve dans la classe animal et qui devrait être abstraite.
- Il faut avoir cette méthode dans la classe crier () **pour pouvoir utiliser le polymorphisme.**

Interface

- Les interfaces sont des techniques utiles pour avoir un code le plus générique possible.

- Elles ne contiennent **que des méthodes abstraites** : aucun attribut d'instance.
- C'est donc un cas particulier de classe abstraite.
- La différence est qu' **une classe peut « implémenter » une ou plusieurs interfaces** alors que l'héritage à tout intérêt à être unique pour éviter les incohérences (et est unique en Java).
- Ca l'oblige donc à écrire le code des méthodes abstraites qui se trouvent dans l'interface.

- C'est une **notion complexe** qui permet de mettre en œuvre au mieux le **polymorphisme** et d'avoir un **code facile à faire évoluer**.

UML - Exercices

Principes

- Pour tous les exercices, il d'agit de créer des classes avec un constructeur et d'écrire le main en utilisant ces classes. On précise bien les paramètres des méthodes.
- On fait le diagramme de classes avec le constructeur et les paramètres des méthodes.

Jeu vidéo

1)

- Soit un jeu vidéo avec des joueurs. Les joueurs ont un nom, une santé, une force et une expérience. Ces trois dernières caractéristiques sont des entiers positifs ou nuls. L'expérience vaut 0 quand le personnage débute.
- On se dote d'un constructeur qui permet de donner un nom, une santé et une force au joueur.
- On peut afficher les caractéristiques d'un joueur (son nom et la valeur de tous ses attributs).
- Un joueur peut attaquer un autre joueur. Une attaque réduit la santé du personnage attaqué de 1. Quand la santé vaut 0, le joueur est mort. Une attaque augmente l'expérience de l'attaquant de 2.
- Écrire un main qui déclare deux joueurs et affiche leur caractéristiques (le deuxième à une santé de 3). Ensuite le main fait attaquer le deuxième joueur par le premier. Afficher les nouvelles caractéristiques des joueurs. Ensuite le premier attaque 2 fois le second. La méthode attaquer affichera le fait que le joueur 2 est mort.

2)

- On crée maintenant des ennemis. Les ennemis ont les mêmes caractéristiques que les joueurs mais ils n'ont pas d'expérience et ils ont une race (troll, elfe, sorcier, etc.), et une valeur qui est un entier positif. Quand un joueur attaque un ennemi, il gagne en expérience la valeur de l'ennemi. Quand un ennemi attaque un personnage, il lui fait perdre 1 point de santé.
- Ecrire un *main* qui déclare deux joueurs et un ennemi et affiche leurs caractéristiques. Ensuite le main fait attaquer l'ennemi par le premier joueur. Puis le main fait attaquer le deuxième joueur par l'ennemi. Afficher les nouvelles caractéristiques de l'ennemi et des joueurs.

Point en couleur

1)

- On définit un point dans un repère géométrique par ses coordonnées x et y.
- On veut pouvoir créer plusieurs points et afficher leurs coordonnées.

2)

- La classe permettant de faire ça existant, on veut pouvoir gérer des points en couleur, sans modifier la première partie Un programme permettra de créer 2 points en couleur et d'afficher leurs coordonnées et leurs couleurs.

Figures géométriques

- On veut gérer des figures géométriques : un rectangle, un carré et un cercle.
- Un cercle est défini par son rayon et son centre qui est un point avec 2 coordonnées x et y
- Le rectangle est défini par son centre et la taille d'un côté horizontal et celle d'un côté vertical. On considère que les rectangles sont forcément posés à l'horizontal.
- Le carré est défini par son centre et la taille de son côté.
- A noter qu'un carré est un rectangle particulier.
- On peut afficher les figures ou les effacer (les rendre invisible). On peut aussi afficher les points ou les effacer.
- On peut récupérer les sommets des carrés et des rectangles qui sont des points.
- On peut déplacer les figures : pour ça on fournit les coordonnées d'un nouveau centre
- On peut changer la taille des figures. Pour cela, une méthode reçoit un coefficient multiplicateur. S'il est compris entre 0 et 1, la figure diminue de taille. S'il est supérieur à 1, la figure augmente de taille.
- Ecrire un programme qui crée un cercle, un rectangle et un carré. Afficher ces figures.
- Afficher les informations de chaque figure.
- Déplacer le rectangle et afficher ses nouvelles informations.
- Affichez les informations des sommes du rectangle après déplacement.

3 : PRINCIPES DE P.O.O.
D'APRES MORELLE
ET DESIGN PATTERN TETE LA PREMIERE

1 – Les 5 concepts fondateurs de la programmation objet

- La POO est basée sur **5 concepts fondateurs** :

1. L'objet (encapsulation : les attributs sont accessibles via les méthodes)

2. Message (l'échange entre deux objets : l'utilisation des méthodes)

L'échange de message est une formulation qui n'est pas intuitive.

Envoyer un message à un objet veut dire : « utiliser une méthode d'un objet ».

Pour utiliser une méthode, on part d'un objet existant.

Par exemple, si dans une méthode m1(), un objet o1 instancie un objet o2 et exécute la méthode m2 : o2.m2(), m2() est un message envoyé par o1 à o2. Il y a alors un échange entre les 2 objets.

On peut concrétiser la situation en imaginant que o1 est une personne « p1 » et que o2 est une personne « p2 ». « p1 » fait quelque chose : il exécute sa méthode m1(). Cette méthode va consister à « créer » « p2 » (c'est-à-dire à le faire venir à lui) puis à demander à « p2 » de faire m2() qu'il sait faire.

3. Classe (la généralisation de l'objet)

4. Héritage (la factorisation de propriétés et de fonctions entre 2 classes)

5. Polymorphisme

2 - Principes de programmation structurée classique appliquées en POO

Évitez la duplication du code

- Tout ce qui se répète doit être factorisé !

Recherchez une cohésion forte = limitez la taille et la complexité

- Une fonction, une méthode, une classe, une bibliothèque, un package doivent avoir une unité logique forte.
- **La cohésion est forte** quand chaque fonctions, méthodes, classe, bibliothèque, package a des **responsabilités bien définis.**
- Une méthode ne doit pas être trop longue :

  ⇒ Une méthode est trop longue dès qu'elle fait plus qu'une tâche logique.
- Une classe ne doit pas être trop complexe :

  ⇒ une classe est trop complexe dès qu'elle ne correspond plus à une entité logique.

Découplez entrées, traitements et sorties : MVC

- Il faut séparer la saisie, le calcul et l'affichage.
- Ça rejoint la séparation M – V – C

  ⇒ M = modèle = données = saisie

  ⇒ V = vue = affichage

  ⇒ C = contrôleur = calcul

Cherchez à faciliter les changements localisés, c'est-à-dire l'évolution du code

- La localisation du changement est un des buts recherchés dans toute programmation.
- Elle sera le produit d'un couplage faible et d'une forte cohésion.

3 - Principes de base de la POO

- Les 2 principes de bases de la POO sont l'encapsulation et la substitution.

Encapsulation

- Le principe d'encapsulation concerne la visibilité :
  - ⇒ Il s'agit de choisir à bon escient quelles seront les méthodes qui seront publiques.

Substitution

- Le principe de substitution met en œuvre plusieurs notions :
  - ⇒ Héritage
  - ⇒ Redéfinition
  - ⇒ Polymorphisme
  - ⇒ Abstraction

4 - L'art du programmeur

- **Eviter tout ce qui peut rendre l'état de l'objet incohérent** et tout ce qui peut correspondre à une évolution incohérente de l'objet.

  ⇒ Incohérent veut dire que l'objet porte des caractéristiques, d'état ou de comportement, qui ne lui appartiennent pas, qui le dénature.

- **L'encapsulation au sens restreint** consiste à éviter un accès direct aux attributs pour éviter que le programmeur puisse leur donner des valeurs sans faire de vérification de cohérence.
- **L'encapsulation au sens large** consiste ce que les actions réalisées par les méthodes le soient de façon cohérente, c'est-à-dire en prenant en compte l'état de l'objet et les règles de fonctionnement de l'objet. Le but est d'éviter toute possibilité d'action incohérente (par exemple : change la pâte d'une gaufre, ou changer la garniture d'une gaufre).
- L'art du programmeur, c'est de gérer proprement l'encapsulation pour offrir un jeu de méthodes judicieux qui protège contre une utilisation non conforme.

5 - Principes avancés de la POO

Encapsulez ce qui varie, que ce soit des attributs ou des méthodes

- En général, on encapsule les attributs.

  ⇒ Il faut aussi penser à encapsuler les méthodes si celles-ci varient en se regroupant en famille.

- Il faut se poser la question suivante :

  ⇒ le changement est-il bien encapsulé ou entrainera-t-il de nombreuses modifications dans le code ?

- L'encapsulation des méthodes se fait via :

  ⇒ des méthodes abstraites

  ⇒ placées dans des classes abstraites ou dans des interfaces.

Préférez la composition à l'héritage – Utiliser l'héritage à bon escient

- **L'héritage** est une relation « **est_un** ».
- **La composition** est une relation « **a_un** ».
- Dans une hiérarchie d'héritage, les objets dérivés doivent avoir une relation « est_un » avec leur classe de base.
- **L'héritage à bon escient concerne en premier lieu les attributs**.
- Pour l'héritage des méthodes, on préférera faire porter les méthodes à la classe mère via une composition d'interface spécifiée à l'instanciation de l'objet dérivé. Ca permet de factoriser des méthodes spécifiques identiques.

  ⇒ Cf. Design Patten « Stragégie ».

Couplez faiblement, autant que possible, les objets qui interagissent

- Une classe dépend d'une autre classe quand elle y fait référence.
- Le couplage entre 2 classes est l'ensemble des références entre les 2 classes.
- Il faut limiter le couplage au maximum en factorisant le plus possible les références.
- Le couplage est faible quand il est limité au maximum à une dépendance avec une interface.
  - ⇒ Dépendez des abstractions. Ne dépendez pas des clases concrètes.
  - ⇒ Coupler faiblement, c'est donc coupler au maximum avec des interfaces. L'instanciation concrète oblige toutefois le plus souvent à dépendre aussi d'une classe concrète.

Programmez des interfaces et non des implémentations

- Autrement dit, adoptez une conception dirigée par les responsabilités.
- L'attribution des bonnes responsabilités aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet : on verra ça dans la méthode GRASP.
- La programmation dirigée par les responsabilités :
  - ⇒ attribue de responsabilités bien définies à chaque classe,
  - ⇒ contribue à réduire le couplage.

4 : GRASP : UNE METHODE POUR BIEN CONCEVOIR CLASSES ET METHODES

Présentation

GRASP

- General Responsibility Assignment Software Pattern
  - ⇒ Pattern d'affectation des responsabilités générales
  - ⇒ Ca veut dire : une méthode pour trouver les méthodes de nos classes !

Responsabilité

- Une responsabilité en P.O.O. c'est une méthode publique d'une classe : un service attendu.
  - ⇒ Une classe a des responsabilités.
  - ⇒ Une méthode publique est une responsabilité.
  - ⇒ Un cas d'utilisation n'est pas une responsabilité.

- Il faut découvrir les responsabilités des classes : ce qu'on attend qu'elle rende comme service pour être utile.
  - ⇒ Attention : les responsabilités ne sont pas les simples « getter » et « setter ».

Objectifs de la méthode GRASP

- Réduire le décalage entre la représentation humaine du problème et sa représentation informatique.

Comment on fait ?

- Penser la conception :
  - ⇒ En termes d'objet proches du réel.
  - ⇒ En termes de responsabilités pour les classes.
  - ⇒ En termes de collaboration concrète entre les objets.

La Collaboration entre objets

- La collaboration entre objet c'est quand une méthode d'un objet O1 utilise une méthode d'un autre objet 02 :
  - ⇒ On dit aussi que O1 envoie un message à O2.
  - ⇒ Quand on trouve une collaboration, on trouve la classe de O2 et une responsabilité de O2.

Les types de responsabilités

La responsabilité FAIRE

- Un objet peut FAIRE c'est-à-dire :
  - ⇒ Accomplir une action (un calcul, une instanciation, etc.)
  - ⇒ Déléguer à un autre objet
  - ⇒ Coordonner des actions (c'est ce que fait un contrôleur).

La responsabilité SAVOIR

- Un objet peut SAVOIR (avec ses « getter »), c'est-à-dire :
  - ⇒ Connaitre les valeurs des attributs : getName(), getAge(), etc.
  - ⇒ Connaitre les objets simples qu'il contient.
  - ⇒ Connaitre les listes qu'il contient.

Principes d'analyse des responsabilités

1 - Principe du spécialiste (expert)

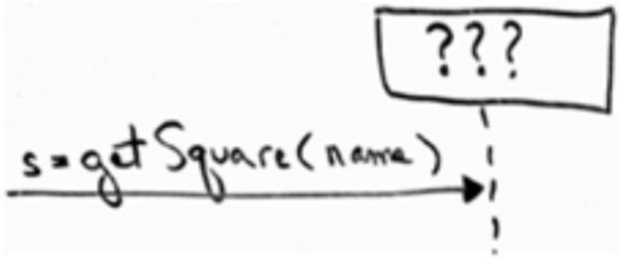
Le problème : qui porte la responsabilité ?

La réponse :

- La classe qui porte les informations manipulées par la responsabilité et particulièrement celles qui sont retournées par la méthode.
- Autrement dit : la classe spécialiste de l'information.

Exemple : le monopoly

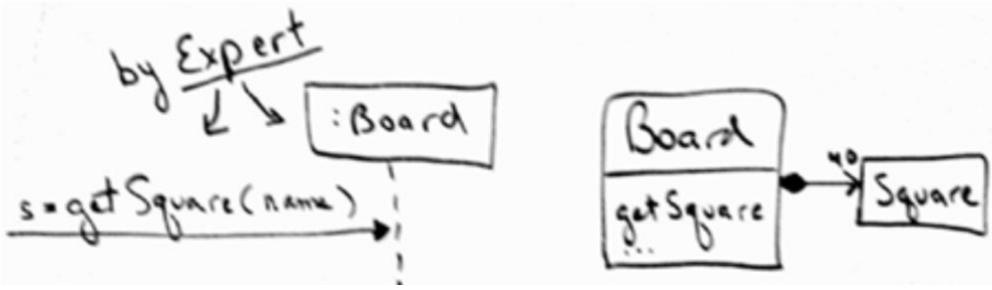
- Qui est responsable de l'accès à une case (un square en anglais) du jeu ?
- Représentation UML du problème :



⇒ La flèche est un appel de méthode sur un objet ???.

⇒ L'appel est écrit : s=getSquare(name)

- Ca suppose un objet « square » et une classe Square. Mais le « square » est un composant du plateau de jeu, un objet « board » :



⇒ :Board : ça veut dire un objet anonnyme de la classe Board

⇒ Board est composé de 40 Square

2 - Principe du créateur (creator)

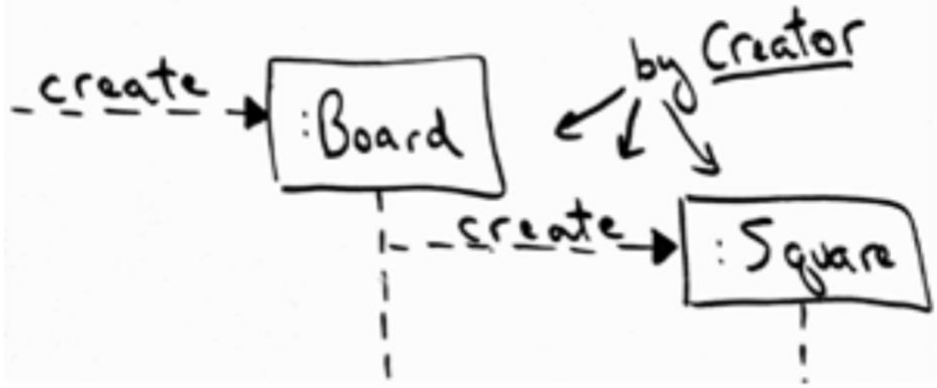
Le problème : qui porte la responsabilité de créer une nouvelle instance ?

La réponse :

- La classe qui contient l'objet comme attribut
- Autrement dit : la classe spécialiste de l'information (on en revient au cas précédent).

Exemple : le monopoly

- Qui est responsable de créer un plateau et ses cases ?



⇒ La flèche pointe directement sur l'objet et pas sur sa ligne de vie : c'est une instanciation.

⇒ L'instanciation d'un « board » instancie des « square ».

3 - Principe du faible couplage

Définition

- Une classe dépend d'une autre classe quand elle y fait référence.
- Le couplage entre 2 classes est l'ensemble des références entre les 2 classes.
- Il faut limiter le couplage au maximum en le factorisant le plus possible les références.

  ⇒ Ce sera un principe de design pattern.
- Le couplage est faible quand il est limité au maximum à une dépendance avec une interface.
- Couplez faiblement, c'est donc coupler au maximum avec des interfaces. L'instanciation concrète oblige toutefois le plus souvent à dépendre aussi d'une classe concrète.

Pourquoi minimiser les dépendances ?

- Pour faciliter les changements.
- Pour faciliter la réutilisation.

Comment minimiser les dépendances ?

- Dépendez des abstractions. Ne dépendez pas des clases concrètes.
- Préférer la transitivité par délégation au lien direct.

Exemple de code

```
mA(){
IB b = new B()
b.mB1()
}
```

⇒ mA() est une méthode de la classe A.

⇒ Cette méthode instancie un objet de la classe B. Donc A dépend de B.

⇒ Mais on instancie un objet de l'interface IB.

⇒ Donc toutes les méthodes appelées par « b » sont définies dans IB.

4 - Principe de la forte cohésion

Définition

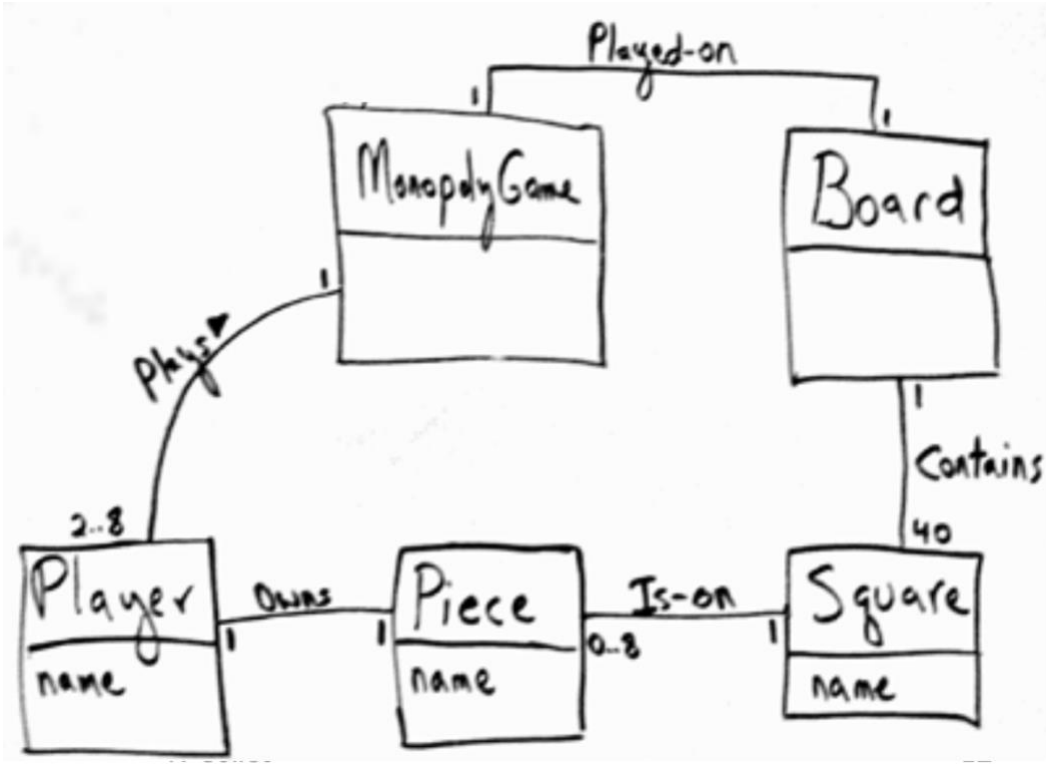
- Une classe de forte cohésion a un petit nombre de méthodes.
  - ⇒ Elle ne fait pas trop de travail.
  - ⇒ Elle fait un travail de même nature.
  - ⇒ On peut la décrire en une seule phrase.
    - ⇒ Ce sera un principe de design pattern.

5 - Principe du contrôleur

Définition

- Un contrôleur coordonne des actions.
- Il joue le rôle d'un « main » ou d'un « sous-main ».
- Le contrôleur participe du fait d'avoir des objets « realiste »
- Un contrôleur porte des méthodes qui correspondent à des cas d'utilisation.

Exemple : ciagramme de classes « réaliste » :



⇒ La classe MonoolyGame c'est une « partie de monopoly » : elle a un plateau avec ses cases et ses pièces sur le jeu. La partie a des joueurs (de 2 à 8). Chaque joueur a 1 pion et 1 pion correspond à un joueur. Un pion est sur une case. Sur une case on peut avoir de 0 à 8 pion.

⇒ Le « player » est un contrôleur de la pièce. Un cas d'utilisation c'est : jouer()

⇒ La « monopolygame » est un contrôleur des joueurs et du plateau. Un cas d'utilisation c'est : installerLaPartie()

6 - Principe du polymorphisme

Définition

- Le polymorphisme, c'est d'avoir un comportement différent pour une même méthode selon l'objet de départ.
- Ca facilite l'écriture du code en évitant les tests sur les types d'objet.

Gains

- Ca permet de créer de nouveaux objets et de les ajouter au code existant facilement :
- Le nouvel objet est utilisé à partir de son interface et n'intervient qu'au moment de son instanciation.

Exemple du monopoly

- Les cases pourraient être une classe abstraite ou une interface dont hériteraient différents type de cases.
- Le contrôleur participe du fait d'avoir des objets « realiste »
- Un contrôleur porte des méthodes qui correspondent à des cas d'utilisation.

Design Patten GoF

On reviendra sur ça avec les DP-GoF.

7 - Principe de pure invention

Problème

- Les objets du monde réels ne sont pas forcément suffisant pour modéliser les objets du logiciel.

Solution

- Il faut créer des objets de toute pièce à forte cohésion et faible couplage.