2 Paradigmes de programmation : impératif, fonctionnel, objet. Exemples et applications.

Paradigme, quèsaco ? Programmer c'est dur. Il faut réfléchir à ce qu'on fait faire à l'ordi, et organiser sa réflexion. Un paradigme c'est un mode de pensée.

I Paradigme impératif

1) Modèle de calcul : machine RAM

La machine a un état interne avec plusieurs composantes :

- ▷ la mémoire programme stocke les instructions à effectuer
- ▷ la mémoire de travail stocke les données. Y compris les registres
- ▷ le compteur d'instruction indique où on en est dans le programme

↳ cycle d'exécution : lire l'instruction actuelle, modifier l'état en fonction de cette instruction, mettre à jour le compteur d'instruction

C'est très proche des vrais ordinateurs

⇒ Paradigme adapté aux programmes interactifs (dont réseau, système, mais aussi UI, jeux...)

2) Instructions typiques, invariants et variants

```
load REG1, #42        load REG2, (REG3)
add REG4, REG1, REG3
```

Haut niveau : `y ← tab[3*x][2]`

```
    load REG1, #tab
debut:
    load REG3, (REG1)
    add REG2, REG2 REG3
    add REG1, REG1 1
    jump_not_eq REG1, #Fintab, debut
```

```
i = 0;
while (i < taille)
{
    sum += tab[i];
    i += 1;
}
```

Pour donner du sens à un programme, il faut raisonner sur le contenu des variables, de la mémoire, aux moments où les instructions s'exécutent.

↳ Un invariant est une propriété (logique) qui est vraie au cours de l'exécution du programme.

Un variant de boucle est une valeur (nombre ou objet + complexe) qui décroît strictement à chaque tour de boucle et est minoré.

Exemples : "$sum = \sum_{k=0}^{i} tab[i]$" est un invariant, et (taille − i) un variant

3) Pile d'appel

On veut réutiliser du code à plusieurs endroits du programme.

Procédure : portion de code qui finit par un instruction de saut (jump) dont la destination peut changer. But : retourner d'où on venait avant.

↳ Convention : la destination est au sommet d'une pile partagée.

les procédures peuvent s'imbriquer indépendamment

- □ D'autres conventions et pratiques (passages de paramètres ; valeur de retour ; en assembleur, protection des registres ; variables locales, etc)

II Paradigme fonctionnel

1) Définition, tranparence référentielle, récursivité

Beaucoup de problèmes se formulent bien en termes mathématiques, plus déclaratifs. La traduction est laborieuse, nuit à la lisibilité, source de bugs...

En programmation fonctionnelle, on construit des fonctions par assemblage (composition, application, etc) de fonctions, et toutes les variables sont <u>immuables</u>: leur valeur est fixée à l'initialisation

Les fonctions vérifient la <u>transparence référentielle</u>: leur valeur de retour ne dépend que des valeurs de leurs paramètres.

```
let rec fibonacci (n : int) : int =
  if n <= 1 then 1
  else fibonacci (n-1) + fibonacci (n-2)
# fibonacci 7;;   → 21
```

Une fonction peut être définie par sa valeur sur d'autres paramètres, elle est alors dite <u>récursive</u>

2) Ordre supérieur, currysfication

Les fonctions elles-même sont des valeurs manipulables:

- ▷ Comme paramètre : une fonction de tri peut être paramétrée par la fonction de comparaison à utiliser entre les éléments
- ▷ Comme valeur de retour
- ▷ Les deux

```
let sens (inverse : bool) : int -> int =
  if inverse then (fun x -> -x) else (fun x -> x)
```

```
List.map (fonct : 'a -> 'b) (l : 'a list) : ('b list)
↳ transforme [x, y, z, ...] en [fonct(x), fonct(y), fonct(z), ...]
List.filter (test : 'a -> bool) (l : 'a list) : ('a list)
↳ garde exactement les élément qui vérifient la fonction test
List.fold_left (f : 'a -> 'b -> 'a) (init : 'a) (l : 'b list) : 'a
↳ transforme [x; y; ...; z] en f(... fff(init, x), y), ...), z)
```

<u>fonctionnelles standard de manipulation des listes en OCaml</u>

<u>Currysfication</u>:

```
let curry (f : ('a * 'b) -> 'c) : 'a -> ('b -> 'c) =
  fun x -> (fun y -> f (x, y))
```

↳ Permet d'appliquer partiellement des fonctions

<u>exemple</u>: (List.fold_left max infinity) : (float list) -> float

3) Équivalence impératif-fonctionnel

- ▫ On peut transformer un programme fonctionnel en impératif
  - ▷ Développement: compilation de fonctions d'ordre supérieur
- ▫ On peut faire apparaître un état interne comme paramètre des fonctions, et donc faire de l'impératif dans du fonctionnel.
  - ↳ une fonction "pure" est une qui sera indépendante de cet état.

↳ Intérêt du fonctionnel : plus facile de raisonner, prouver. Plus stable (en prog concurrente, etc).

III Paradigme objet

1) Motivation, interface

Pour des gros projets, on veut compartimenter, modulariser le code.
En programmation orientée objet, on définit des briques de construction, en séparant la façon dont elles se comportent vu de l'extérieur, qu'on appelle <u>interface</u>, de leur mécanique interne, ou <u>implémentation</u>. Le comportement d'un objet dépend de sa <u>classe</u>.

Idée : pouvoir développer, tester, déboguer, optimiser chaque classe, sans se soucier de l'implémentation des autres classes utilisées.

Une classe est composée : ▷ d'<u>attributs</u>, les données représentant l'état interne de chaque objet de la classe

▷ de <u>méthodes</u>, les procédures qui dépendent d'un objet et le font évoluer.

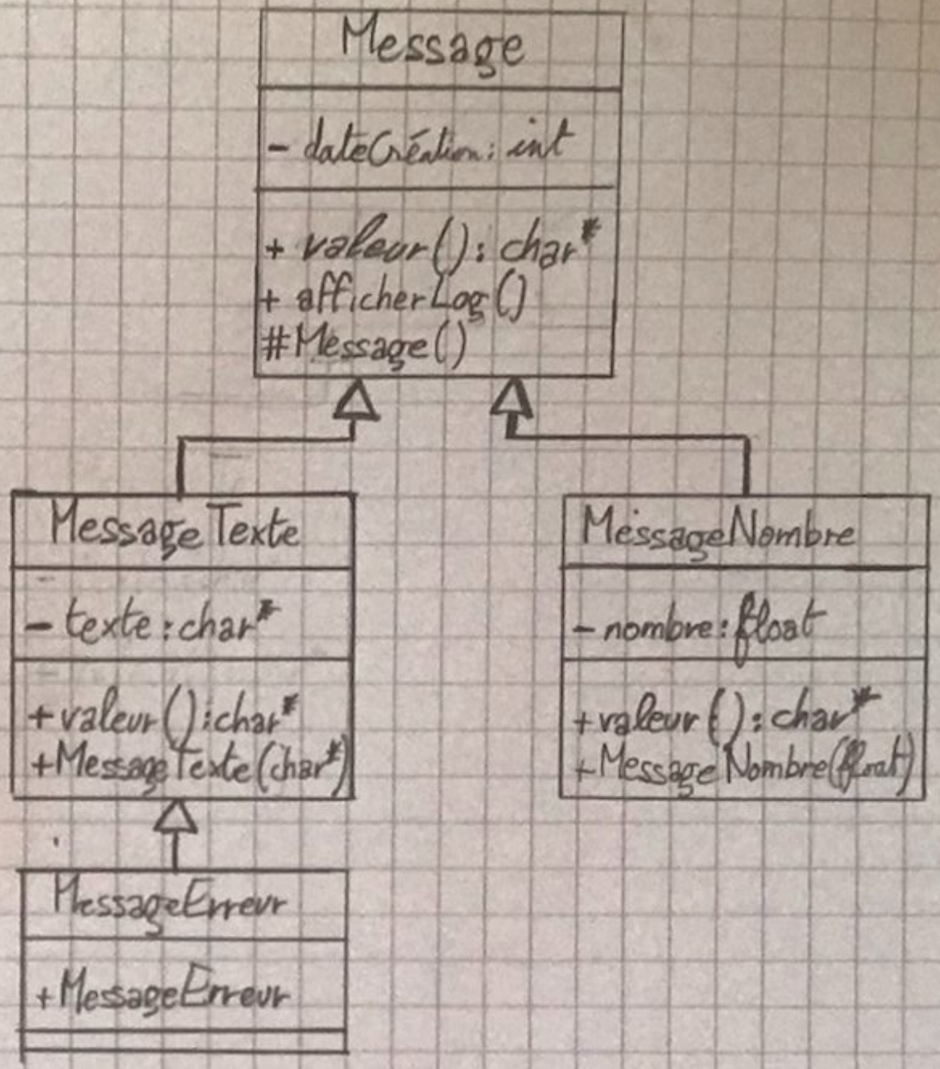
↳ L'interface est composée des types des méthodes et d'une description (plus ou moins formelle) de leurs effets de bord.

2) Héritage, méthodes virtuelles

Principe : une classe qui est une spécialisation d'une autre doit pouvoir <u>hériter</u> de son implémentation et de son interface.

Utilités : ▷ réutilisation de code

▷ <u>polymorphisme</u> : plusieurs implémentations spécialisées différemment peuvent être utilisées de façon interchangeable.



Une méthode <u>virtuelle</u> est une méthode qui peut être redéfinie dans une spécialisation.

▷ Développement : implémentation de méthodes virtuelles en C