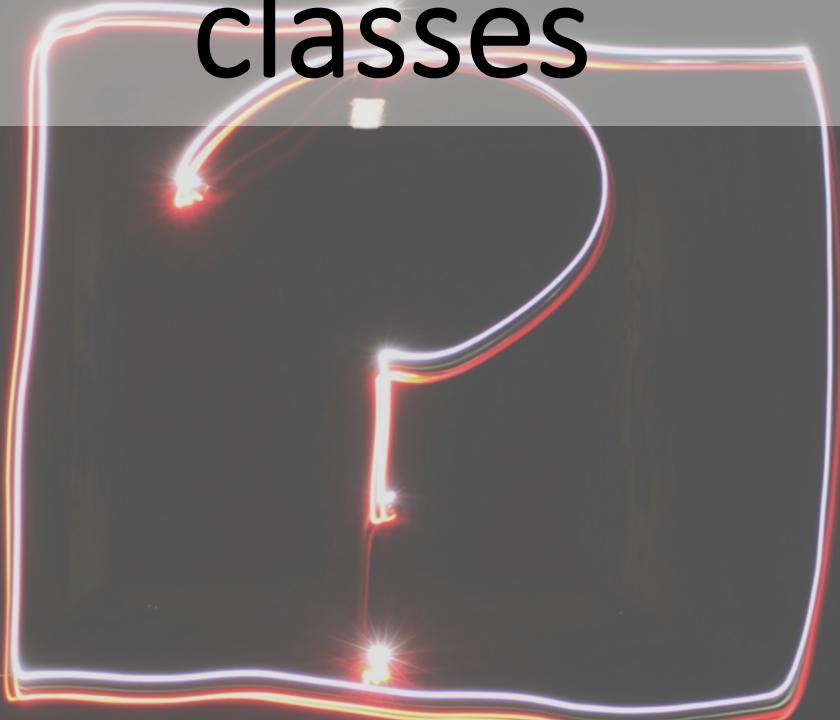


Quelques règles de design de classes



Objectifs

- Rappels sur les principes :
 - DRY / WET
 - KISS
- Comprendre les lois de Démeter
- Comprendre le « Tell don't ask »

DRY (Don't Repeat Yourself)

- **Buts** : faciliter la maintenance, améliorer la lecture du code, la testabilité et le coût
- Le code ne devrait pas être répété : **attention aux copier/coller**
- **Bien pire encore** : la connaissance ne doit pas être dupliquée (ex. règles d'affaire)
- C'est une façon de respecter le SRP de **SOLID**

Violation du DRY

- WET : We enjoy typing (ou Wasting Everyone's Time ou Write Everthing twice)
- Si le code est dupliqué et si un modification est nécessaire, le code doit être changé dans tous les endroits où ce dernier a été copié

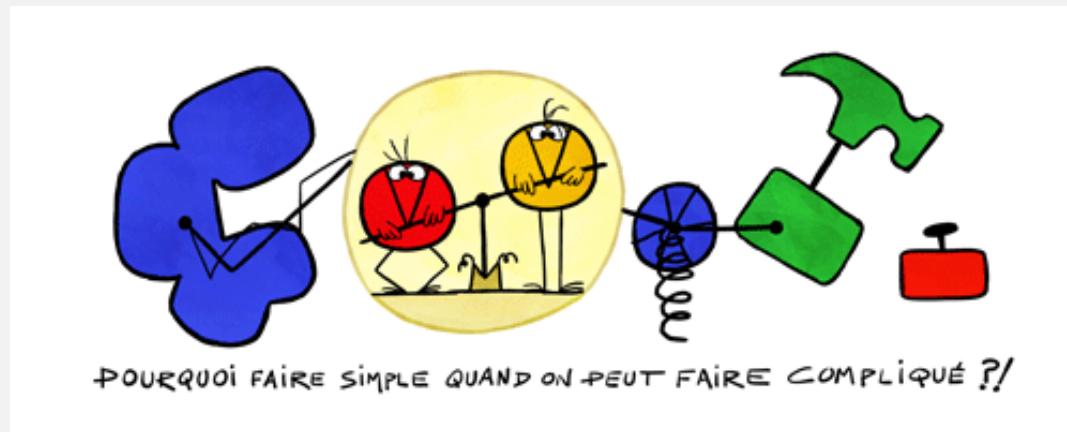
⇒Code difficile à maintenir

```
12
13  if num1 == 0 and sign == '+' and num2 == 0:
14      print("0+0 = 0")
15  if num1 == 0 and sign == '+' and num2 == 1:
16      print("0+1 = 1")
17  if num1 == 0 and sign == '+' and num2 == 2:
18      print("0+2 = 2")
19  if num1 == 0 and sign == '+' and num2 == 3:
20      print("0+3 = 3")
20814     print("50*47 = 2350")
20815  if num1 == 50 and sign == '*' and num2 == 48:
20816      print("50*48 = 2400")
20817  if num1 == 50 and sign == '*' and num2 == 49:
20818      print("50*49 = 2450")
20819  if num1 == 50 and sign == '*' and num2 == 50:
20820      print("50*50 = 2500")
20821
20822  print("Thanks for using this calculator, goodbye :))")
```

* Code extrait de : https://github.com/AceLewis/my_first_calculator.py/blob/master/my_first_calculator.py (meme)

KISS et dérivés

- Keep It Simple, Stupid
 - Ou... *Keep it small and simple*
 - *Keep it sweet and simple*
- Le code doit être simple à lire et à comprendre



Les Shadoks

```
int n = 0;
int y = 1;

do
{
    n = Console.In.ReadInt();
    Console.Out.WriteLine();
} while (n <= 0);

for (int i = 1; i <= (n * (n + 1)) / 2; i++)
{
    Console.Out.Write('*');
    if (i == (y * (y + 1)) / 2)
    {
        Console.Out.WriteLine();
        y++;
    }
}
y = y - 2;

for (int i = 1; i <= (n * (n - 1)) / 2; i++)
{
    Console.Out.Write('*');
    if (i == (n * (n - 1)) / 2 - (y * (y - 1)) / 2)
    {
        Console.Out.WriteLine();
        y--;
    }
}
```

KISS et dérivés

- « **La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer** » (**Antoine de Saint-Exupéry**)
- Éviter les trop longues fonctions
- Ne mélangez pas les différents niveaux d'abstraction

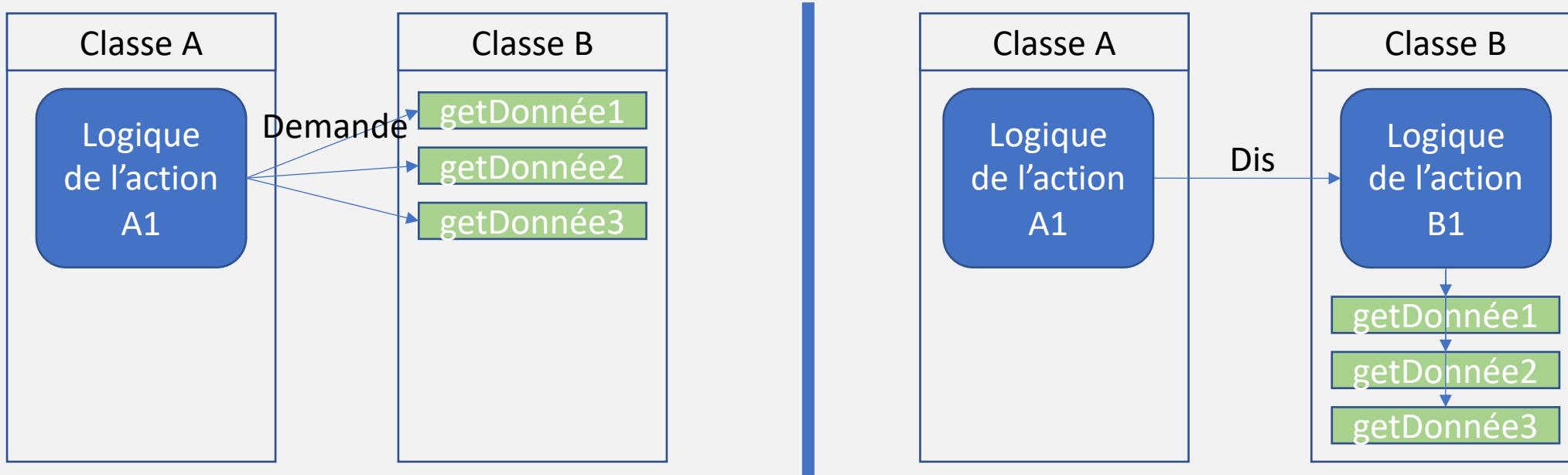
```
public class TraitementDonnees
{
    // [...]
    public ResultatTraitement TraiterDonnees()
    {
        Donnees donnees = SourceDonnees.ObtenirDonnees();
        Donnees donneesNormalisees = NormaliseurDonnees.Normaliser(donnees);
        Donnees donneesFiltrees = FiltreDonnees.Filtrer(donneesNormalisees);
        ResultatTraitement resultat = AlgorithmeTraitement.Executer(donneesFiltrees);

        return resultat;
    }
}
```

Processus
Procédure
[...]
Opérations arithmétiques, fichiers, réseau, etc.

Tell don't ask

- Les objets interagissent par échange de messages (ou appels d'actions)
- Selon ce principe, on ne demande pas des renseignements à un objet : on lui demande d'effectuer une action



Tell don't ask

- En programmation procédurale, on interroge des structures de données
- En programmation orientée objet, on dit aux objets quoi faire
 - ⇒ Le comment ne regarde que l'objet
 - ⇒ Ne pas respecter ce principe revient à violer le principe d'encapsulation
 - ⇒ Éviter les « getters » (Penser en termes de capacités pas de données à représenter)
 - ⇒ **Attention, le message n'est pas de les interdire**

Loi de Démeter

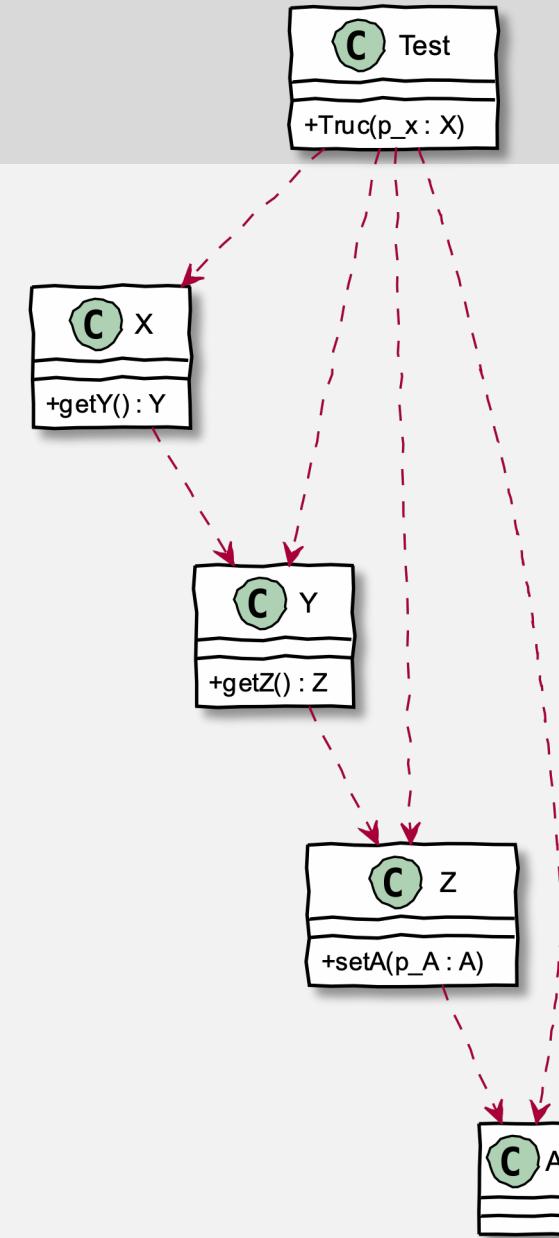
- Chaque élément ne doit parler qu'à ses voisins immédiats
- Chaque élément ne doit parler qu'à ses amis et ne pas parler aux étrangers
- Ne parler qu'à ses amis directs
- N'en dites pas trop !

Plus formellement, la Loi de Déméter pour les fonctions requiert que toute méthode M d'un objet O peut simplement invoquer les méthodes des types suivants d'objets :

1. O lui-même
2. les paramètres de M
3. les objets que M crée/instancie
4. les objets membres de O

Exemple

```
public class Test
{
    public void Truc(X p_x)
    {
        A a = new A();
        p_x.getY().getZ().setA(a);
    }
}
```



A -----> B : A dépend de B

Au final... Faut-il toujours appliquer ces principes ?

- Ce sont des principes, pas des obligations
- Ils sont là pour améliorer la structure d'une application, d'un système
- Ces principes s'appliquent bien aussi en architecture de systèmes
- Le but est d'améliorer la maintenabilité des applications

