Top-down design

DESIGN

- Magie noire !
- Méthodes usuelles :
  - **Top-down (de haut en bas)** : de la tache vers le détail
  - Bottom-up (de bas en haut) : des détails vers la tâche
  - Orientation objet : pilotage par les donnée

Top-down design dans l' idéal

Idéalement, nous n'avons pas besoin d'extraire des fonctions du code existant. Notre code est bien dès la première fois !

- Écrire la **fonction principale** comme un appel à des fonction de support
- Écrire une implémentation (vide) pour chacun de ces **fonctions de support**
- Ensuite, **concevoir le code** de chaque fonctions de support comme un appel à des sous fonctions de support...etc

On appelle cela la décomposition hiérarchique et/ou le top-down design

Dans la pratique :

- Mais en pratique on use de techniques de design mixtes
- On refactore (= on restructure le code) fréquemment

Différentes étapes :

- Définir le problème (ça peut prendre une page entière)
- Décider logiquement comment cela devrait marcher
  - Hiérarchiquement : partir du problème général et affiner continuellement vers les détails
- Traduire le 'programme logique' en code
  - Les détails en **instruction**
  - Les instructions liées en f**onctions**
  - Les fonctions liées en **modules**
- Refactorer le code pour améliorer le design

Exemple : le jeu du morpion

Séquence de jeu

Pseudocode du top-level

1. Faire un plateau
2. Déterminer si l'utilisateur veut un 'O' ou un 'X'
3. Jouer au jeu jusqu'à une victoire ou un match nul
4. Afficher le résultat

Code en Python du top-level :

```
def main():
     """Programme pour jouer au jeu du
     Morpion"""
    plateau = nouveau_plateau()
    joueur_humain = demande_O_ou_X()
    resultat = joue_jeu(joueur_humain,
                        plateau)
    affiche_resultat(resultat)
```

Analyse du top-level

- **2** et **4** sont des pénibles (mais facile) entrée/sorties
- **1** concerne la principale structure de donnée (important)
  - Une possibilité : Liste de liste

```
Plateau = [['.','.',''],
           ['.','.','.'],
           ['.','.','.']]
```

- On peut écrire maintenant ces fonctions...

- **3** est plus complexe et nécessite une décomposition hiérarchique (top-down)

3. Jouer au jeu jusqu'à une victoire ou un match nul

Pseudocode affiné du 3 (level 2)

Jouer au jeu jusqu'à une victoire ou un match nul

```
Tant que l'on joue encore :
    3.1 Afficher le plateau
    3.2 Jouer un tour pour celui qui
        doit jouer
    3.3 Vérifier si il y a une
        victoire ou un match nul
```

Pseudocode affiné du 3.2 (level 3)

Jouer un tour pour celui qui doit jouer

```
Si humain_tour:
   humain_place(plateau)# 3.2.1
Sinon:
   ordinateur_place(plateau)# 3.2.2
```

Pseudocode affiné du 3.2.1 (level 4)

humain_place(plateau)

```
Tant que not choix_val:
     position = obtient_position()
    choix_val=pos_libre(plateau,position)
     Si not choix_val:
          Ecrire message d'erreur
```

Pseudocode affiné du 3.2.2 (level 4)

ordinateur_place(plateau)

- Peut être difficile
- Pour l'instant :
  - choisir au hasard une case vide

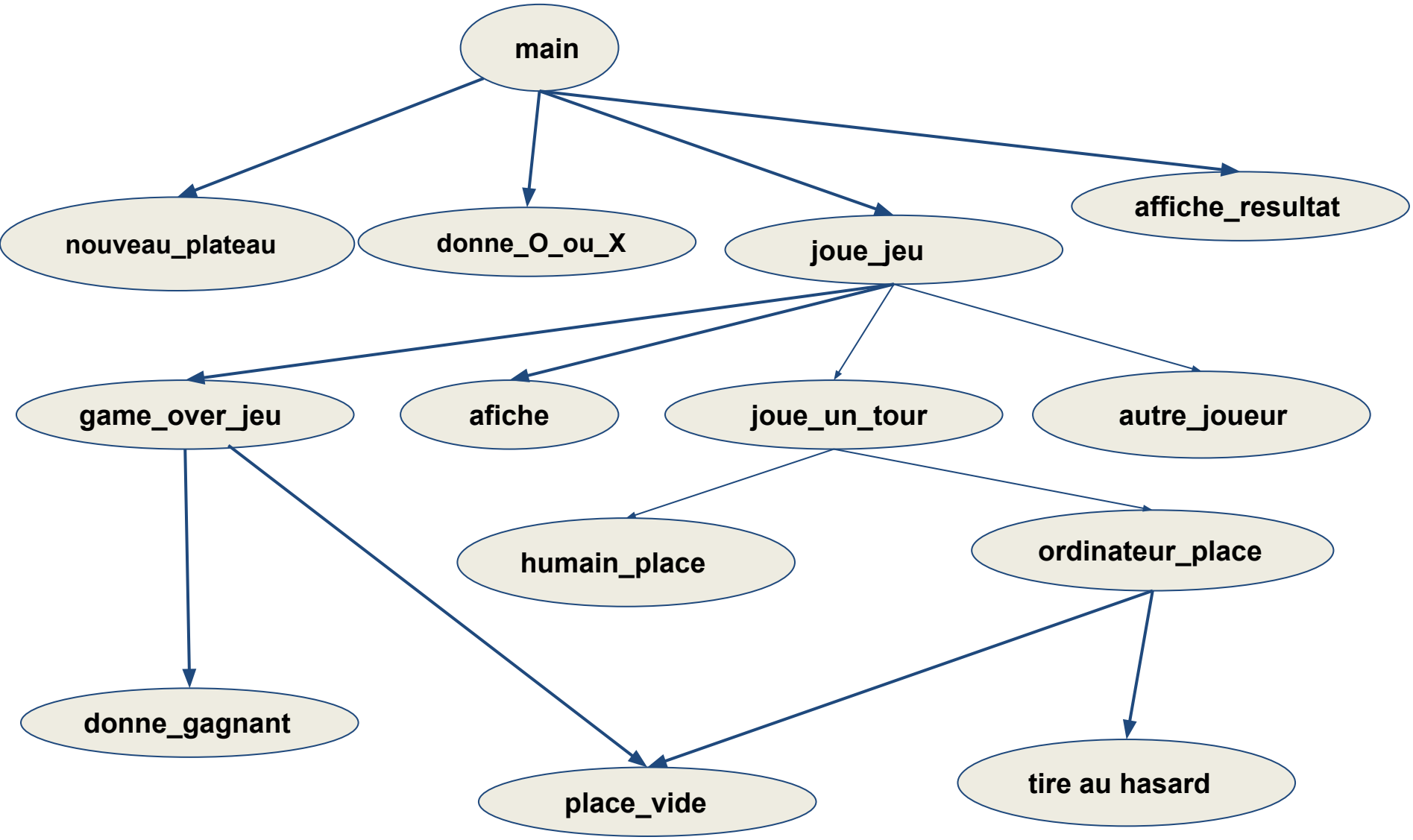
Pseudocode affiné du 3.3 (level 3)

Vérifier si il y a une victoire ou un match nul

```
 Retourne vrai si un joueur à 3
symbole alignés ou  si le plateau
est plein
```

Vue d'ensemble

On continue jusqu'à ce que tout soit implémenté:



Conclusion : Ce que vous pouvez avoir appris

La programmation est :

- Créative
- 'Explorative'
- Expérimentale

Le design n'est pas un processus formel

Il faut s'attendre à faire des erreurs : cela peut être des découvertes

Les fonctions sont la clé d'un bon design

Elles permettent de

- 'Diviser pour régner '(abstraction)
- Tester le code
- Confiner le code