# TP Chapitre 3 : Interface et Implémentation

**Objectifs du TP**

**Objectifs du TP.** À l'issue de cette séance, tu dois être capable de :

— distinguer l'**interface** d'une structure de données (le *quoi* : nom des méthodes, paramètres, comportement attendu) de son **implémentation** (le *comment* : réalisation interne) ;

— spécifier une structure de données par son interface, sans écrire le code des méthodes ;

— écrire la définition d'une classe, accéder à ses attributs et à ses méthodes ;

— implémenter une interface en Python : la classe `Tableau`, la classe `Pile`, la classe `File` ;

— traduire un pseudo-code en Python en n'utilisant que l'interface d'une structure de données ;

— écrire plusieurs implémentations d'une même structure de données : la file avec une liste, puis avec deux piles ;

— comparer le coût des opérations (ajouter en fin de liste ou insérer au début) et choisir une implémentation adaptée.

**Consignes générales.** Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

## Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation.

### Exercice A1 — La classe Tableau du cours (interface et implémentation)

Au chapitre, on a fixé l'interface du tableau par quatre méthodes : `ajout(index, element)`, `suppr(index)`, `est_vide()` et `longueur()`. La classe `Tableau` ci-dessous est une *enveloppe* : elle expose cette interface et stocke les éléments dans l'attribut `data`, de type liste. Pour tester la classe, on utilisera les prix d'articles d'un ticket de caisse.

```
class Tableau:
    def __init__(self):
        self.data = []

    def ajout(self, index, element):
        # A COMPLETER : inserer element dans data a la position index
        ...

    def suppr(self, index):
        # A COMPLETER : retirer de data l'element situe a la position index
        ...

    def est_vide(self):
        # A COMPLETER : renvoyer True si data est vide, False sinon
        ...

    def longueur(self):
        # A COMPLETER : renvoyer le nombre d'elements contenus dans data
        ...

```

```
tab = Tableau()
tab.ajout(0, 2.5)
tab.ajout(1, 1.2)
tab.ajout(0, 3.0)
print(tab.est_vide())
print(tab.longueur())
```

**Point de validation**

Ce programme doit afficher `False` puis `3`. Après les trois insertions, `tab.data` vaut `[3.0, 2.5, 1.2]` : on a inséré 3,0 € en tête, les prix 2,5 € et 1,2 € ayant été décalés d'un cran.

### Exercice A2 — La classe Pile : le tas de tickets de caisse (mode LIFO)

À la fin de la journée, le gérant empile les tickets de caisse (les montants) les uns sur les autres. Le dernier ticket posé sur le tas est le premier repris : c'est le mode **LIFO** (*Last In, First Out*), qui caractérise la structure de données **pile**. On spécifie l'interface : `empiler(element)` ajoute un ticket en haut de la pile ; `depiler()` retire et renvoie le ticket en haut de la pile ; `est_vide()` teste si la pile est vide ; `longueur()` compte les tickets. Compléter l'implémentation : les tickets sont stockés dans l'attribut `data`, de type liste.

```
class Pile:
    def __init__(self):
        self.data = []

    def empiler(self, element):
        # A COMPLETER : ajouter element en haut de la pile
        ...

    def depiler(self):
        # A COMPLETER : retirer et renvoyer l'element en haut de la pile
        ...

    def est_vide(self):
        # A COMPLETER : renvoyer True si la pile est vide, False sinon
        ...

    def longueur(self):
        # A COMPLETER : renvoyer le nombre d'elements contenus dans la pile
        ...

pile = Pile()
pile.empiler(12.5)
pile.empiler(3.9)
pile.empiler(8.0)
print(pile.depiler())
print(pile.longueur())
print(pile.est_vide())
print(pile.depiler())
```

**Point de validation**

Ce programme doit afficher `8.0`, puis `2`, puis `False`, puis `3.9`. Le dernier ticket empilé (8,0 €) est dépilé en premier : c'est le mode LIFO.

### Exercice A3 — Du pseudo-code au Python : le total du panier

La caisse doit additionner les prix des articles d'un panier. En pseudo-code, en utilisant l'interface du tableau (`longueur` et `element`) :

```
total = 0
pour i de 0 a longueur(tableau) - 1 :
    total = total + element(tableau, i)
renvoyer total
```

L'interface du tableau définie au chapitre ne permet pas encore de consulter un élément : on l'enrichit d'abord avec la méthode `element(index)`, qui renvoie l'élément situé à l'index donné. Compléter la méthode `element`, puis la fonction `total_panier` qui implémente le pseudo-code en n'utilisant *que* l'interface de la classe `Tableau`.

```
class Tableau:
    def __init__(self):
        self.data = []

    def ajout(self, index, element):
        self.data.insert(index, element)

    def suppr(self, index):
        del self.data[index]

    def est_vide(self):
        return len(self.data) == 0

    def longueur(self):
        return len(self.data)

    def element(self, index):
        # A COMPLETER : renvoyer l'element situe a la position index
        ...


def total_panier(tableau):
    total = 0
    # A COMPLETER : pour i de 0 a longueur(tableau) - 1 :
    #   total = total + element(tableau, i)
    return total
```

**Point de validation**

Après avoir inséré 12,5 €, 3,9 € puis 2,0 € dans un tableau (aux positions 0, 1 puis 2), `total_panier(tableau)` doit renvoyer `18.4`. La classe complétée doit rester compatible avec le programme de l'exercice A1 (`False` puis `3`).

### Exercice A4 — Le coût des opérations : ajouter en fin ou insérer au début

La caisse enregistre les prix des articles d'une longue file de clients. On veut comparer deux façons de construire une liste de prix : ajouter chaque prix en fin de liste avec `append`, ou l'insérer au début avec `insert(0, x)`. Le programme ci-dessous construit les deux listes et mesure le temps de chaque construction avec `time.perf_counter()`. Compléter les deux boucles, puis lancer le programme.

```
import time

# 12 000 prix de tickets simules
prix = [2.5, 1.2, 3.0, 4.8, 0.9] * 2400
```

```
5
6  # Version 1 : ajout en fin de liste
7  liste_fin = []
8  t0 = time.perf_counter()
9  for p in prix:
10     # A COMPLETER : ajouter p en fin de liste_fin
11 t1 = time.perf_counter()
12 print("Ajout en fin :", t1 - t0, "secondes")
13
14 # Version 2 : insertion au debut de liste
15 liste_debut = []
16 t0 = time.perf_counter()
17 for p in prix:
18     # A COMPLETER : inserer p en debut de liste_debut
19 t1 = time.perf_counter()
20 print("Ajout au debut :", t1 - t0, "secondes")
```

Avant de lancer le programme, réponds à la question : que doit faire Python avec les éléments déjà présents dans la liste quand on insère au début ? (rappel du cours : une liste Python est un tableau contigu en mémoire.)

**Point de validation**

L'insertion au début est nettement plus lente que l'ajout en fin (sur la machine de référence, environ 37 fois plus lente ; en général plusieurs dizaines de fois pour 12 000 prix). Les deux listes contiennent les 12 000 prix : `liste_fin` est identique à `prix`, `liste_debut` en est l'ordre inverse.

## Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification, réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites.

### Exercice B1 — Le porte-monnaie électronique (spécifier une interface, puis l'implémenter)

Un client paie ses achats avec un porte-monnaie électronique : une carte prépayée qui contient un solde en euros. On veut modéliser ce porte-monnaie par une classe `Portefeuille`. On commence par *spécifier* l'interface :

- `__init__(montant_initial)` : crée un porte-monnaie avec le solde `montant_initial` ;
- `crediter(montant)` : ajoute `montant` au solde (ne renvoie rien) ;
- `payer(montant)` : si le solde est suffisant, retire `montant` du solde et renvoie `True` ; sinon renvoie `False` *sans modifier* le solde ;
- `consulter_solde()` : renvoie le solde courant ;
- `est_vide()` : renvoie `True` si le solde est nul, `False` sinon.

Écrire la classe `Portefeuille` complète (initialisation et méthodes), puis la tester avec le scénario de l'encadré vert.

**Spécification.**

- *Entrées* : le solde initial (flottant, en euros) ; les montants des crédits et des paiements.

— *Traitement* : la classe stocke le solde dans l'attribut `solde` ; `payer` ne débite que si le solde est suffisant.

— *Sorties* : les valeurs renvoyées par les méthodes : `True` ou `False` pour `payer`, le solde pour `consulter_solde`, `True`/`False` pour `est_vide`.

— *Contraintes* : un seul attribut ; aucun `print` dans les méthodes ; le solde ne doit jamais devenir négatif.

**Pour t'aider — questions de conception (sans donner le code)**

— Quels attributs la classe doit-elle avoir ? Quel est leur type ?

— Pourquoi `payer` doit-il renvoyer `True` ou `False` ? Quel est l'intérêt pour le programme qui utilise la classe ?

— Pourquoi aucune méthode ne doit-elle afficher de texte avec `print` ? (que doit renvoyer `consulter_solde` ?)

— Pourquoi ne peut-on pas nommer à la fois l'attribut et la méthode « solde » ? (essaye, puis corrige)

— Comment garantir que le solde ne devient jamais négatif ?

**Point de validation**

Avec le programme : `p = Portefeuille(50.0)` ; `p.crediter(20.0)` ; `print(p.payer(30.0))` ; `print(p.consulter_solde())` ; `print(p.payer(100.0))` ; `print(p.consulter_solde())` ; `print(p.est_vide())`. Affichages attendus : `True`, `40.0`, `False`, `40.0`, `False`.

### Exercice B2 — La file d'attente de la caisse (implémentation avec une liste)

Au supermarché, les clients font la queue à la caisse : un nouveau client se place en fin de file et le caissier sert le client qui est en tête de file. C'est le mode **FIFO** (*First In, First Out*), qui caractérise la structure de données **file**. On modélise la file par une classe `FileListe` dont l'interface est : `enfiler(element)` ajoute un client en fin de file ; `defiler()` retire et renvoie le client en tête de file ; `est_vide()` teste si la file est vide ; `longueur()` compte les clients en attente.

**Énoncé.** Écrire la classe `FileListe` complète : les clients sont stockés dans l'attribut `data`, de type liste ; `enfiler` ajoute en fin (`append`) et `defiler` retire en tête (`pop(0)`).

**Spécification.**

— *Entrées* : une suite de clients (chaînes de caractères).

— *Traitement* : `enfiler` : `append(element)` en fin de liste ; `defiler` : `pop(0)` retire l'élément en tête de liste.

— *Sorties* : `defiler` renvoie le client servi ; `est_vide` renvoie un booléen ; `longueur` renvoie le nombre de clients en attente.

— *Contraintes* : mode FIFO ; seules les méthodes `append` et `pop` de la classe `list` sont autorisées ; on ne doit jamais appeler `defiler` sur une file vide.

**Pour t'aider — questions de conception (sans donner le code)**

— Quelles méthodes de la classe `list` réalisent l'enfiler et le défiler ?

— Dans quel ordre les clients sont-ils servis ? (teste mentalement avec Ali, Baya, Chen)

— Que se passe-t-il si on appelle `defiler` sur une file vide ? Comment l'éviter ?

— (question pour la suite) Que doit faire `pop(0)` avec les clients déjà présents dans la liste ?

**Point de validation**

Avec le programme : `f = FileListe()` ; `f.enfiler("Ali")` ; `f.enfiler("Baya")` ; `f.enfiler("Chen")` ; `print(f.defiler())` ; `print(f.longueur())` ; `print(f.defiler())` ; `print(f.est_vide())` ; `print(f.defiler())` ; `print(f.est_vide())`. Affichages attendus : `Ali`, `2`, `Baya`, `False`, `Chen`, `True`.

### Exercice B3 — La même file, avec deux piles

On veut une *autre implémentation* de la même interface : la file est réalisée avec **deux piles** (la classe `Pile` de l'exercice A2, que tu recopies dans ton fichier). L'idée : on enfile dans la pile `entree` ; pour défiler, si la pile `sortie` est vide, on vide la pile `entree` dans la pile `sortie` — le premier client arrivé se retrouve alors en haut de la pile `sortie` — puis on dépile la pile `sortie`.

**Énoncé.** Écrire la classe `FileDeuxPiles` avec les attributs `entree` et `sortie` (deux objets `Pile`) et les méthodes de l'interface : `enfiler`, `defiler`, `est_vide`, `longueur`. Interdiction d'accéder directement à l'attribut `data` des piles : on n'utilise que leur interface (`empiler`, `depiler`, `est_vide`, `longueur`).

**Spécification.**

— *Entrées* : une suite de clients (chaînes de caractères).

— *Traitement* : `enfiler` = empiler dans `entree` ; `defiler` : si `sortie` est vide, transférer tous les éléments de `entree` vers `sortie` (dépiler de `entree` puis empiler dans `sortie`, dans une boucle `while`), puis dépiler de `sortie`.

— *Sorties* : `defiler` renvoie le client servi ; `est_vide` renvoie `True` si les deux piles sont vides ; `longueur` renvoie `longueur(entree) + longueur(sortie)`.

— *Contraintes* : on n'utilise que l'interface de la classe `Pile` ; jamais d'accès à `.data` ; le même programme de test que l'exercice B2 doit donner les mêmes affichages.

**Pour t'aider — questions de conception (sans donner le code)**

— Pourquoi faut-il deux piles ? Que contient chacune d'elles ?

— Pourquoi ne transfère-t-on l'entrée vers la sortie que lorsque la sortie est vide ? (que se passerait-il sinon ?)

— Pourquoi la méthode `longueur` doit-elle additionner les longueurs des deux piles ?

— Pourquoi le programme de l'exercice B2 doit-il fonctionner sans modification avec `FileDeuxPiles` ? (c'est tout l'intérêt d'une interface)

**Point de validation**

Avec exactement le même programme que l'exercice B2 (en remplaçant seulement `FileListe` par `FileDeuxPiles`), les affichages doivent être identiques : `Ali`, `2`, `Baya`, `False`, `Chen`, `True`.

### Exercice B4 — Comparer les coûts sur des données de caisse

Le supermarché hésite entre les deux implémentations de file. On veut mesurer le coût réel de chacune sur un grand nombre de clients.

**Énoncé.** Écrire un programme qui :

— demande le nombre de clients $N$ (entier supérieur ou égal à 1 000) ;

— crée une `FileListe`, y enfile $N$ clients (« Client 1 », « Client 2 », …), retire et affiche le premier client, puis défile les $N - 1$ clients restants ;

— chronomètre l'ensemble (enfilements + défilements) avec `time.perf_counter()` ;

— recommence à l'identique avec `FileDeuxPiles` ;

— affiche les deux temps, le rapport (temps liste ÷ temps deux piles) et le nom du premier client servi par chaque implémentation.

**Spécification.**

— *Entrées* : $N$, le nombre de clients (entier $\geq 1000$).

— *Traitement* : pour chaque implémentation : $N$ enfilements puis $N$ défilements ; chronométrage avec `time.perf_counter()` avant le premier enfilement et après le dernier défilement.

— *Sorties* : le temps de `FileListe`, le temps de `FileDeuxPiles`, le rapport des deux temps, et le premier client servi par chaque implémentation.

— *Contraintes* : `import time` ; même nombre d'opérations pour les deux implémentations ; le premier client est retiré *avant* la boucle des défilements restants.

**Pour t'aider — questions de conception (sans donner le code)**

— Pourquoi le premier client est-il retiré avant la boucle ? (réfléchis à ce que contiendrait la variable « premier » à la fin d'une boucle de $N$ défilements)

— Quand la file contient $k$ clients, combien d'éléments `pop(0)` doit-il décaler ?

— Pourquoi `FileDeuxPiles` est-il globalement plus rapide ? (chaque client n'est transféré qu'une seule fois de l'entrée vers la sortie)

— Pourquoi les deux programmes doivent-ils afficher le même premier client ?

**Point de validation**

Avec $N = 100\,000$, le temps de `FileDeuxPiles` est nettement inférieur à celui de `FileListe` (le rapport est généralement supérieur à 10 ; sur la machine de référence, environ 20). Les deux implémentations affichent `Client 1` comme premier client servi.

## Partie C — Exercice de réflexion (à finir à la maison)

**À finir à la maison**

**La matinée de caisse du supermarché.**

**Contexte.** Un supermarché ouvre sa caisse à 8 h. Les clients arrivent un par un, à raison d'un client par minute. Chaque client a un panier : un nombre d'articles (entre 1 et 30) et un prix moyen de 3 € par article. Le caissier sert un client à la fois ; chaque article met 3 secondes à être enregistré.

**Objectif.** Écrire un programme qui simule la matinée (par exemple 120 minutes) et réponde à des questions comme : combien de clients ont été servis ? Combien de clients attendent encore à la fermeture ? Quel est le temps d'attente moyen ? Quelle est la longueur maximale de la file ? Quelle est la recette totale ? Quelle implémentation de file choisir pour une journée à 10 000 clients ?

**Questions à se poser avant de coder.**

— Quelles sont les données d'entrée ? (durée de la matinée, temps de service par article, prix moyen par article, nombre d'articles de chaque client)

— Comment modéliser un client ? (deux informations : son heure d'arrivée et son nombre d'articles — un couple `(heure, articles)` suffit)

— Comment organiser le service ? (si la caisse est libre — fin du service précédent atteinte — et la file non vide, on sert le client en tête : durée du service = articles × 3 secondes)

— Pourquoi une file (FIFO) et pas une pile pour la file d'attente ?

— Comment calculer le temps d'attente d'un client ? (heure de début de service − heure d'arrivée)

— Comment vérifier le programme ? (penser à un petit cas : 3 clients espacés de 2 minutes avec un article chacun → aucune attente)

— Comment passer d'une implémentation de file à l'autre sans tout réécrire ? (changer une seule ligne : le nom de la classe)

**Étapes suggérées.**

1. Version 1, sans file : un client arrive chaque minute, la caisse est toujours libre ; vérifier la recette à la main (articles × 3 €).
2. Version 2, avec `FileListe` : gérer la file d'attente ; vérifier sur le petit cas de 3 clients.
3. Version 3 : statistiques — afficher clients servis, clients restants, attente moyenne, longueur maximale, recette.
4. Version 4 : changer une seule ligne pour `FileDeuxPiles` : les statistiques doivent être identiques ; mesurer les temps d'exécution pour 10 000, 50 000, 100 000 et 200 000 clients et observer le rapport.
5. Bonus : augmenter la fréquence d'arrivée (un client toutes les 30 secondes) : que devient la file d'attente ? Et si le prix moyen passait à 4 € par article ?

**Contraintes.** Données crédibles (articles entre 1 et 30, prix entre 1 € et 5 € par article) ; programme testé et commenté ; les messages peuvent être écrits sans accents (ex. `File`, `attente`, `recette`) si ton éditeur pose problème ; on n'accède jamais à l'attribut `data` en dehors des classes.

*Fin du TP — la partie C est à finir à la maison.*