Corrigé Activité P1C2

Pour modéliser les différents types d'utilisateurs de ChessMaster Pro, l'approche POO est idéale car elle permet de gérer la spécialisation et les comportements spécifiques de manière organisée.

1. Héritage pour la structure de base

Nous allons utiliser l'**héritage** pour définir une hiérarchie claire.

- **Classe Mère : Utilisateur**
  - Cette classe contiendra les attributs communs à tous (e.g., identifiant, email, motDePasse).
  - Elle contiendra également les méthodes génériques (e.g., seConnecter(), seDeconnecter()).
- **Classes Filles : Administrateur, Joueur, Spectateur**
  - Chacune hérite automatiquement des propriétés de base de Utilisateur.
  - Elles peuvent ensuite ajouter leurs propres attributs spécifiques (e.g., Joueur ajoute classementElo).

Justification : L'héritage assure la **réutilisabilité** du code de base et permet la **spécialisation** de chaque rôle.

2. Encapsulation pour la sécurité

Tous les attributs (comme l'email et surtout le motDePasse) de la classe Utilisateur seront rendus **privés**.

- Nous fournirons des méthodes publiques (getEmail(), setMotDePasse(nouveauMP)).
- Le *setter* de motDePasse ne se contentera pas de stocker la valeur brute; il devra appliquer un algorithme de hachage pour des raisons de sécurité.

Justification : L'encapsulation **protège l'état interne** de l'objet, limitant l'accès direct au mot de passe et garantissant qu'il est toujours stocké sous une forme sécurisée, assurant ainsi la **cohérence** des données.

3. Polymorphisme pour les permissions

Supposons que chaque utilisateur ait une méthode afficherMenu(). Le contenu du menu dépend du rôle.

- La classe mère Utilisateur pourrait avoir une méthode afficherMenu() par défaut.
- La classe fille Administrateur **redéfinira** cette méthode pour afficher en plus les options de gestion (création de tournois, bannissement de joueurs).
- La classe Joueur redéfinira la méthode pour mettre en avant l'accès aux parties en cours et aux statistiques.

Justification : Le **polymorphisme** permet au système de traiter n'importe quel objet, qu'il soit Administrateur ou Joueur, comme un simple Utilisateur et d'appeler utilisateur.afficherMenu(). Le comportement spécifique s'exécutera automatiquement au moment de l'exécution (*runtime*). Cela rend le code d'appel très propre et flexible face à l'ajout de nouveaux rôles futurs.

Exemple de Code conceptuel (Python/Pseudocode)

Python

```
// Classe Mère

class Utilisateur:

     def __init__(self, email, mdp_hash):

// Attributs privés

     self.__email = email

     self.__motDePasse = mdp_hash # Déjà hashé par le
constructeur

// Méthodes d'instance de base

    def seConnecter(self):

          print(f"L'utilisateur {self.__email} se
connecte.")
```

```
// Méthode polymorphe par défaut
     def afficherMenu(self):
        print("Menu Utilisateur : Profil | Aide | Déconnexion")
// Classe Fille spécialisée
     class Joueur(Utilisateur):
         def __init__(self, email, mdp_hash, elo=1200):
               super().__init__(email, mdp_hash)
               self.classementElo = elo
// Redéfinition (Polymorphisme)
    def afficherMenu(self):
        print("Menu Joueur : Jouer une partie | Mes statistiques | Déconnexion")
// Instance et démonstration
     joueur_alice = Joueur("alice@chess.com", "hash_mdp_alice")
     admin_bob = Utilisateur("bob@admin.com", "hash_mdp_bob") // Imaginons qu'Admin n'ajoute pas de méthode supplémentaire pour cet exemple
# L'appel est uniforme (Polymorphisme)
     joueur_alice.afficherMenu() // Affiche : "Menu Joueur : ..."
     admin_bob.afficherMenu()   // Affiche : "Menu Utilisateur : ..."
```

Ce corrigé illustre parfaitement comment la POO (Héritage, Encapsulation, Polymorphisme) permet de créer une structure modulaire, sécurisée et capable d'évoluer si ChessMaster Pro décide d'ajouter un rôle comme "Entraîneur".