

Examen terminal: Programmation

L1 MPC1

28 mai 2026 - Durée: 2h

On rappelle qu'aucun document, ni équipement électrique ou électronique n'est autorisé.

Bien que une Game Boy Color pourra être utilisée sans le son si vous avez besoin de patienter.

Objectif de l'examen Cet examen évalue votre capacité à concevoir et implémenter un programme orienté objet en Python, et à appliquer le *design pattern* MVC (Modèle-Vue-Contrôleur). À travers une application de gestion de bibliothèque, vous serez amenés à définir des classes, à comprendre la séparation des responsabilités, puis à structurer un programme complet en trois composants distincts.

Les exercices de cet examen :

- sont au nombre de 3;
- les exercices ne sont pas indépendants mais peuvent être menés en parallèle;
- sont chacun constitués de plusieurs parties de difficultés croissantes;

RENDEZ DES COPIES SÉPARÉES POUR CHAQUE EXERCICE, CECI VOUS PERMETTRA DE D'Y REVENIR AU COURS DE L'EXAMEN SANS PERDRE LE CORRECTEUR.

Sujet conçu par Claude (et) François.

EXERCICE 1 – PROGRAMMATION ORIENTÉE OBJET

Le contexte de cet exercice est une application de gestion d'une petite bibliothèque personnelle.

1.1 La classe Livre

On vous donne la classe suivante :

```
1 class Livre:
2     def __init__(self, titre, auteur):
3         self.titre = titre
4         self.auteur = auteur
5         self._disponible = True
```

Question 1.1.1 Qu'appelle-t-on un *attribut* ? Listez les attributs d'instance de la classe `Livre` et donnez leur type.

Question 1.1.2 Pourquoi l'attribut `_disponible` est-il préfixé d'un tiret bas ? Ajoutez une méthode de nom `est_disponible` qui rend sa valeur.

Question 1.1.3 Complétez la méthode suivante. Elle doit retourner `True` si le livre était disponible (et le marquer comme emprunté), `False` sinon :

```
1 def emprunter(self):
2     ...
```

Question 1.1.4 Écrivez la méthode `__str__`. Elle doit retourner une chaîne du type :

```
"Trilogie spinoziste (Jean-Bernard Pouy) [disponible]"
```

en indiquant `[disponible]` ou `[emprunté]` selon l'état du livre.

1.2 La classe Bibliothèque

On vous donne le début de la classe suivante :

```
1 class Bibliothèque:
2     def __init__(self):
3         self._livres = []
```

Question 1.2.1 Ajoutez une méthode `Bibliothèque.ajouter(livre)` qui ajoute un livre à une bibliothèque et une méthode `Bibliothèque.livres_disponibles()` qui retourne la liste des livres actuellement disponibles.

Question 1.2.2 Ajoutez une méthode `Bibliothèque.chercher(titre)` qui retourne le `Livre` dont le titre correspond, ou `None` si introuvable.

Puis, en utilisant `chercher`, ajoutez une méthode `Bibliothèque.emprunter(titre)` qui tente d'emprunter le livre portant ce titre et retourne `True` en cas de succès.

1.3 Héritage : le livre numérique

On souhaite modéliser les livres numériques. Contrairement à un livre physique, un livre numérique peut toujours être emprunté : il existe en nombre illimité de copies.

Question 1.3.1 Qu'appelle-t-on l'héritage en programmation orientée objet ? Quelle relation établit-il entre une classe parente et une classe fille ?

Question 1.3.2 Écrivez une classe `LivreNumérique` qui hérite de `Livre`. Son constructeur prendra les mêmes paramètres que celui de `Livre` et appellera le constructeur parent à l'aide de `super()`.

Question 1.3.3 Surchargez la méthode `emprunter` dans `LivreNumérique` de façon à ce qu'elle retourne toujours `True`, sans modifier l'attribut `_disponible`.

Question 1.3.4 Surchargez `__str__` pour retourner une chaîne indiquant que le livre est numérique, par exemple :

```
"Trilogie spinoziste (Jean-Bernard Pouy) [numérique]"
```

Question 1.3.5 Sans modifier la classe `Bibliothèque`, peut-on lui ajouter un `LivreNumérique` via `ajouter` ? La méthode `emprunter` de `Bibliothèque` fonctionnera-t-elle correctement avec ce livre ? Quel principe de la programmation orientée objet illustre ce comportement ?

1.4 Modélisation UML

Question 1.4.1 Dessinez la boîte UML de la classe Livre.

Question 1.4.2 Ajoutez LivreNumérique au diagramme.

Question 1.4.3 Ajoutez Bibliothèque au diagramme.

EXERCICE 2 – SÉPARATION DES RESPONSABILITÉS

On vous donne le code suivant d'une application de bibliothèque écrite sans aucune architecture particulière :

```
1 def gerer_bibliotheque():
2     livres = [] # liste de dict {"titre": str, "auteur": str, "dispo": bool}
3
4     while True:
5         print("1. Livres disponibles  2. Emprunter  0. Quitter")
6         choix = input("Votre choix : ") # (a)
7
8         if choix == "1":
9             dispo = [l for l in livres if l["dispo"]] # (b)
10            for l in dispo:
11                print(f"- {l['titre']} ({l['auteur']})" # (c)
12            elif choix == "2":
13                titre = input("Titre : ") # (d)
14                livre = next((l for l in livres # (e)
15                             if l["titre"] == titre), None)
16                if livre and livre["dispo"]:
17                    livre["dispo"] = False # (f)
18                    print("Emprunté.")
19                else:
20                    print("Indisponible ou introuvable.")
21            elif choix == "0":
22                break
```

2.1 Analyse

Un *design pattern* (ou patron de conception) est une solution générale et réutilisable à un problème récurrent de conception logicielle. Il ne s'agit pas de code prêt à l'emploi, mais d'un modèle qui guide la structure d'un programme.

Question 2.1.1 Le *design pattern* MVC découpe une application en trois composants :

- le **Modèle** gère les données et les règles métier ;
- la **Vue** gère l'affichage et la saisie ;
- le **Contrôleur** coordonne le modèle et la vue.

Pour chacun des éléments (a) à (f) du code ci-dessus, indiquez s'il relève du Modèle, de la Vue ou du Contrôleur. Justifiez brièvement.

Question 2.1.2 Donnez deux inconvénients concrets de ce code monolithique (c'est-à-dire où toute la logique, l'affichage et la gestion des données sont mélangés dans une seule et même fonction) par rapport à une architecture MVC.

Question 2.1.3 Un développeur souhaite remplacer l'interface en ligne de commande par une interface web. Quelles parties du code serait-il obligé de réécrire? Justifiez en vous appuyant sur le MVC.

2.2 Refactoring par composition

Question 2.2.1 Quelle est la différence entre héritage et composition? En vous appuyant sur les classes Livre et Bibliothèque, donnez un exemple de chacune des deux relations.

Question 2.2.2 On souhaite transformer la fonction `gerer_bibliotheque` en une classe `Application` qui utilise par composition une instance de `Bibliothèque`. Écrivez le constructeur de `Application` qui crée et stocke une `Bibliothèque` vide comme attribut.

Question 2.2.3 Transformez le bloc `elif choix == "2":` (lignes 13 à 20) en une méthode `emprunter(self)` de la classe `Application`, en utilisant l'attribut `Bibliothèque` créé à la question précédente.

2.3 Modélisation UML

Question 2.3.1 Dessinez le diagramme de classes UML de la classe `Application` (définie en partie 2) ainsi que son lien avec la classe `Bibliothèque`.

Question 2.3.2 Dans l'architecture MVC, le Contrôleur dépend du Modèle et de la Vue. Représentez schématiquement en UML la différence entre :

- un Contrôleur qui *crée* lui-même le Modèle et la Vue dans son constructeur ;
- un Contrôleur qui *reçoit* le Modèle et la Vue en paramètre de son constructeur.

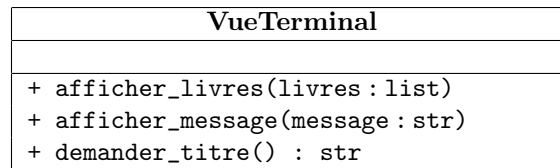
Quelle relation UML correspond à chaque cas ? Quel avantage apporte le second cas ?

EXERCICE 3 – IMPLÉMENTATION DU *DESIGN PATTERN* MVC

On reprend les classes `Livre` et `Bibliothèque` de l'exercice 1.

3.1 La Vue

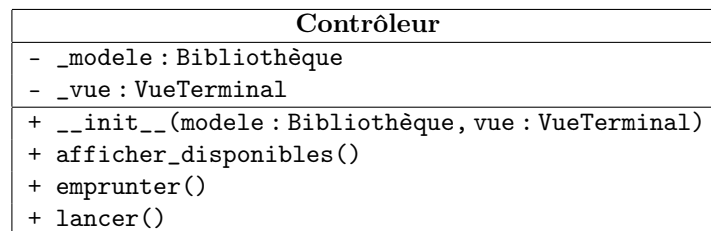
On vous donne le diagramme UML suivant :



Question 3.1.1 Implémentez en Python la classe `VueTerminal`. `afficher_livres` affichera chaque livre sur une ligne en utilisant `str(livre)` ; si la liste est vide, elle affichera "Aucun livre disponible."

3.2 Le Contrôleur

On vous donne le diagramme UML suivant :



Question 3.2.1 Implémentez `__init__`, `afficher_disponibles` et `emprunter`.

`emprunter` doit : demander un titre via la vue, tenter l'emprunt via le modèle, puis afficher un message de succès ou d'échec via la vue.

Question 3.2.2 Implémentez `lancer` qui affiche en boucle le menu suivant et appelle la méthode du contrôleur correspondante :

1. Livres disponibles 2. Emprunter 0. Quitter

3.3 Extensibilité

Question 3.3.1 Écrivez le code de lancement de l'application : créez un modèle, ajoutez-y deux livres, créez une vue et un contrôleur, puis lancez l'application.

Question 3.3.2 On souhaite remplacer `VueTerminal` par une classe `VueGraphique` sans modifier `Contrôleur`. Quelles méthodes `VueGraphique` doit-elle obligatoirement posséder ? Justifiez.

3.4 Modélisation UML

Question 3.4.1 Dessinez le diagramme de classes UML complet de l'architecture MVC implémentée dans cet exercice. Faites apparaître `Contrôleur`, `Bibliothèque` et `VueTerminal` avec leurs attributs, leurs méthodes principales et les relations qui les unissent (nature de chaque relation et multiplicités).

Question 3.4.2 Ajoutez `VueGraphique` au diagramme. Quelle relation entretient-elle avec `VueTerminal` ? Comment le diagramme montre-t-il que `Contrôleur` peut fonctionner indifféremment avec l'une ou l'autre ?