Laboratoire de Programmation en JAVA

BAC2 Bachelier en informatique (toutes orientations)

Année académique 2024-2025

Jean-Marc Wagner
Mounawar Madani
Patrick Quettier

1. Introduction

1.1 Informations générales : UE, AA et règles d'évaluation

Cet énoncé de laboratoire concerne les unités d'enseignement (UE) suivantes :

« Développement Orienté Objet et Multitâche »

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Introduction aux désign patterns et aux tests unitaires** (15h, Pond. 13%)
- **Multi-Threading** (22,5h, Pond. 19%)
- **Programmation orientée objet en Java** (37,5h, Pond. 31%)
- **Programmation orientée objet en C#** (45h, Pond. 37%)

Ce laboratoire intervient dans la construction de la côte de l'AA « Programmation orientée objet en Java ».

La cote de l'AA « Programmation orientée objet en Java » est construite selon :

♦ théorie : un examen écrit en juin 2024 et coté sur 20
♦ laboratoire **(cet énoncé)** : **une évaluation en cours de quadrimestre (35%) et une évaluation pendant la session d'examen (65%) → cote sur 20**
♦ note finale : **moyenne géométrique de la note de théorie (50%) et de la note de laboratoire (50%)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session.

1) Chaque étudiant doit être capable d'expliquer et de justifier l'intégralité du travail.

2) En 2ème session, un **report de note** est possible pour la théorie ou le laboratoire **pour des notes supérieures ou égales à 10/20**. Les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

3) Les consignes de présentation des travaux de laboratoire sont fournies par les différents professeurs de laboratoire.

Consigne importante concernant ChatGPT :

« **ChatGPT** (ou toute aide « IA » à la production automatique de code) est ton ami mais… Pour qu'il le reste, tu devras être capable d'expliquer tout le code généré par **ChatGPT**. Tout code généré par **ChatGPT** que tu seras incapable d'expliquer en détails débouchera sur une cote nulle concernant la question sur ce code (**ET** pour la partie concernée du projet), même si celui-ci est correct et fonctionnel.

Les étudiants devront être en mesure à l'oral d'écrire des petits fragments de code similaire ou d'apporter des modifications mineures sur le code généré »

1.2 Contexte et objectifs

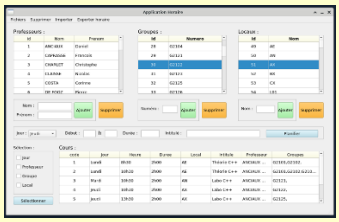
L'objectif de ce laboratoire est de développer une application graphique fenêtrée utilisant les diverses techniques Java vues au cours théorique.

Pour cette application, nous allons faire appel à votre créativité. Vous avez certainement un hobby, un sport, un centre d'intérêt, une thématique pour laquelle vous pourriez créer cette application. Les seuls éléments imposés sont des éléments techniques qui doivent apparaître dans celle-ci et décrits plus loin dans cet énoncé.

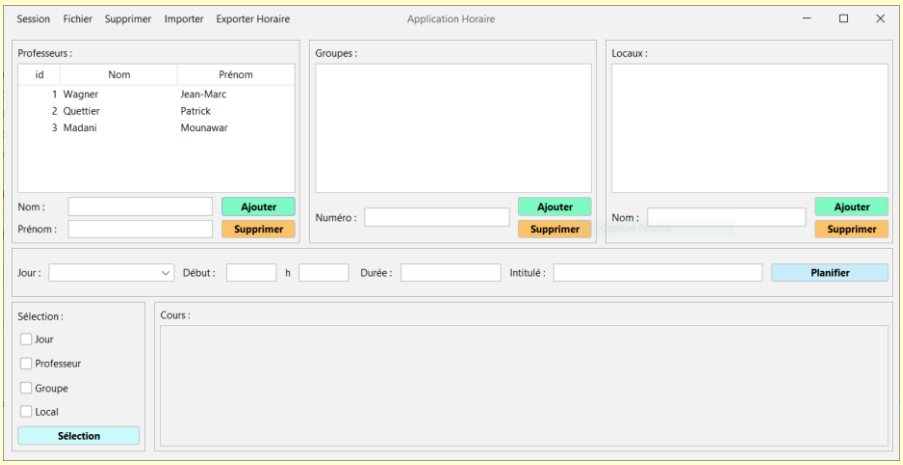
Une première consigne est qu'il doit s'agir d'une **application de gestion de données** présentant

- des entités (les objets « métiers » manipulés par l'application)
- les opérations classiques de type CRUD (create – read – update – delete) sur ces entités
- des fonctionnalités mettant en application les entités ci-dessus.
- Une entrée en session : seuls les utilisateurs disposant d'un login et d'un mot de passe corrects pourront utiliser l'application

Si vous manquez d'inspiration, il vous est possible de vous baser sur l'énoncé du laboratoire de C++ de cette année intitulée « Gestion d'horaire de cours » et de l'étendre pour tenir compte des contraintes techniques imposées. Néanmoins, c'est vous à présent qui allez concevoir toute l'interface graphique.



→



C++ Java

Dans cet exemple, les entités sont les professeurs, les groupes, les locaux et les cours. On retrouve les opérations d'ajout et suppression par l'intermédiaire des boutons verts et oranges. Une des fonctionnalités est la planification de cours (bouton bleu) mais aussi l'importation de fichiers csv et l'exportation de l'horaire au format texte. Dans la barre de menu de la version Java de l'application, on voit apparaître un menu item « Session » qui présentera des items « Login » et « Logout » permettant à l'utilisateur d'entrer en session via une boîte de dialogue permettant d'encoder un login et un mot de passe.

Avant tout développement, la thématique abordée devra néanmoins être validée par votre professeur de laboratoire.

1.3 Environnement de développement et choix du JDK

Tout d'abord, il est nécessaire d'installer le JDK (Java SE Development Kit). On vous demande d'utiliser la **version 21 du JDK**. Vous pouvez directement le télécharger à partir du site d'Oracle :
https://www.oracle.com/java/technologies/downloads/

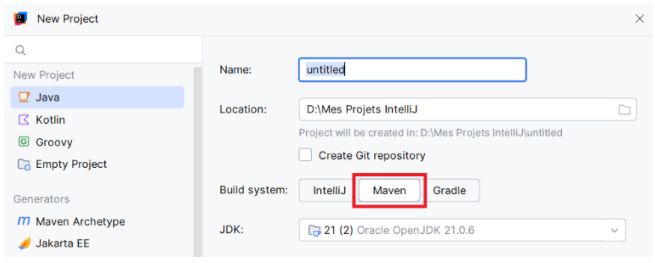
Si vous l'installez sous Windows, choisissez l'installeur (x64 installer) XXX.exe (https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe pour la version 21).

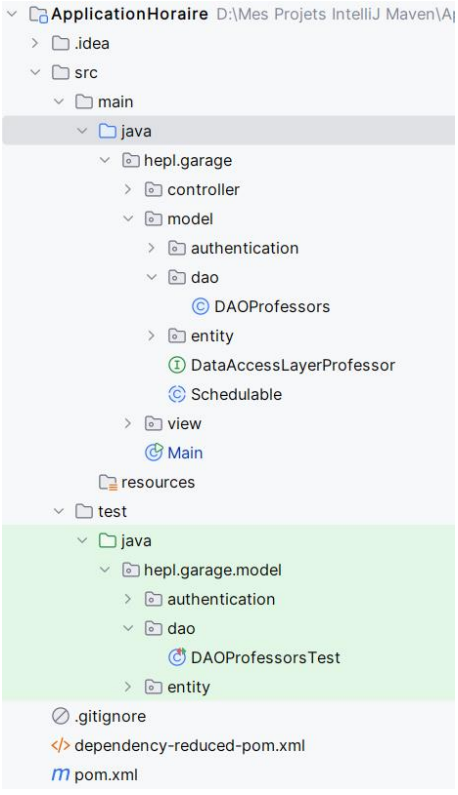
Contrainte : on vous demande d'utiliser l'IDE « **IntelliJ IDEA** » de la société JetBrains. En tant qu'étudiant de la HEPL, vous avez droit à la version « **Ultimate** ». Pour cela, vous devez tout d'abord vous inscrire sur le site de JetBrains afin de vous faire reconnaître en tant qu'étudiant :
https://www.jetbrains.com/shop/eform/students

Ensuite, voici un lien vers le téléchargement si nécessaire :
https://www.jetbrains.com/idea/download/#section=windows

1.3 Structure du projet et tests unitaires

Lors de la création de votre projet, IntelliJ vous demandera le format du projet (« Build System ») :





On vous demande de choisir « **Maven** » qui permettra de mettre en place vos tests unitaires (utilisation de **JUnit**) dans le répertoire « test » prévu à cet effet. Ce répertoire contiendra les packages et classes « miroirs » de vos packages et classes de votre application à tester.

2. Contraintes techniques : les composants de l'application

2.1 Les données

Au minimum,

- **4 classes de données** (les **classes « métiers »** de votre application, encore appelées les **entités**) doivent être créés dont une au moins doit être **abstraite** (certaines devront avoir des relations entre elles : référence d'une dans l'autre, héritage, conteneur de l'une dans l'autre, …). Elles doivent contenir différentes sortes de données → au minimum :
  - **String**,
  - **int (ou Integer)**,
  - **float (ou Float)**,
  - **LocalDate**
  - éventuellement **image** (à vous de voir selon votre thème)
- éventuellement une **interface** implémentée par une ou plusieurs des classes développées ci-dessus.

De plus, d'autres classes/interfaces devront être créées pour les besoins de l'architecture de l'application. Une classe (ou plusieurs classes) en particulier devra servir de **modèle** et **contenir toutes les données de l'application** – ces données seront stockées dans des conteneurs Java (ArrayList, LinkedList, …). Cette classe pourrait être construite sous la forme d'un **singleton de l'application** (→ voir plus loin : « architecture logicielle de l'application »)

IMPORTANT : Toutes vos classes métiers devront respecter au maximum les **principes SOLID**, en particulier ici le principe **S** de **« Single Responsability »**

Cas de l'application « Horaire » : On pourrait imaginer les classes **Schedulable**, **Professor**, **Group**, **Classroom**, **Course**. Schedulable serait une classe abstraite (contenant un id) dont héritent Professor, Group et Classroom. On pourrait imaginer un interface estIdentifiable qui aurait une méthode getId() car les professeurs, les groupes, mais aussi les cours, ont des identifiants qui les identifient de manière unique. La classe Course contiendrait un conteneur de groupes et des références vers le professeur et le local. Toutes les données (professeurs, groupes, …) de l'application seraient stockées au sein de conteneurs dans une classe (ou des classes – les DAO ; voir plus loin) qui ferait office de « model » de l'application.

2.2 Persistance des données et fichiers

Votre application devra manipuler plusieurs types de fichiers :

- Un ou des fichiers binaires (ou encre « **d'objets sérialisés** ») : le contenu des conteneurs de toutes vos entités (le « modèle ») devra être écrit sur disque de cette manière.
- Un ou des **fichiers textes** et/ou **csv** en lecture et/ou écriture

- Un ou des **fichiers properties** manipulés avec la classe Java **Properties** → Un de ces fichiers contiendra les logins et mots de passe des utilisateurs pouvant utiliser l'application → sa manipulation devra se faire à l'aide du **patron de conception « Template Method »** vu par Mr. Madani.

  Un autre exemple de fichier properties pourrait être un fichier de configuration de l'application.

Cas de l'application « Horaire » : Tous les conteneurs de professeurs, groupes, locaux et cours seraient enregistré sur disque sous la forme d'un ou plusieurs objet(s) sérialisé(s) dans un ou plusieurs fichiers. Des fichiers professors.csv, groups.csv et classrooms.csv pourraient être réutilisés et lus pour importer automatiquement des professeurs, des groupes et des locaux dans l'application. On pourrait aussi envisager d'exporter l'horaire d'un Schedulable au format txt. Les login et mots de passe des employés devront être stockés dans un fichier properties (« users.properties ») et manipulé par une instance de la classe Properties. Un fichier Properties pourrait également être utilisé pour gérer les paramètres de l'application (nom des fichiers, …)

2.3 Interface graphique utilisateur

Les interfaces graphiques devront être créées avec **Swing** et être ergonomiques, intuitives, agréables à utiliser et redimensionnables tout en s'adaptant aux dimensions des fenêtres :

- La fenêtre principale (instance d'une classe dérivée de la classe **JFrame**) affichera et manipulera les données : elle comportera au moins
  - une **JTable** (TableModel personnalisé ou pas) → permettant par exemple d'afficher les entités de votre application
  - un ou des **JComboBox**
  - une ou plusieurs **images**
  - un ou plusieurs JTextField, JLabel, …
  - un ou des JButton
- La fenêtre principale devra comporter une barre de menu (instance de la classe **JMenuBar**) disposant d'au moins 2 JMenu ayant chacun plusieurs JMenuItem.
- La fenêtre principale (suite à des clics sur des JButton ou des JMenuItem) devra faire apparaître au moins 2 boîtes de dialogue différentes (instance de classes dérivées de **JDialog**) personnalisées que vos aurez donc créés entièrement vous-même.
- Une boîte de dialogue instance de la classe **JOptionPane** devra également être utilisée.
- L'affichage des dates devraient être paramétrée à l'aide de la classe **DateTimeFormatter**. Le format pourra être modifié en cours d'exécution à l'aide d'un item de menu « Paramètres ».

La gestion des événements sera :

- Soit gérée directement par IntelliJ dans le cas des **événements secondaires** comme un clic de sélection d'une ligne d'une JTable, etc… en fait pour tout événement qui ne donne pas lieu à une modification du « modèle » des données.
- Soit en mettant en place un modèle MVC dans le cas des **événements principaux**, c'est-à-dire donnant lieu à une modification du modèle des données. Dans ce cas, la gestion des

événements sera déléguée à une classe **Controleur** que vous devez créer vous-même et qui implémentera les listeners d'événements provenant de l'interface graphique. Le contrôleur manipulera ainsi le modèle de l'application et la fenêtre principale → void pdf sur « Principes SOLID et architecture MVC »)

Cas de l'application « Horaire » : L'interface graphique de l'application « Horaire » comporte visuellement tous les éléments graphiques demandés. L'entrée en session peut se réaliser à l'aide d'une boite de dialogue personnalisée.

2.4 Architecture logicielle de votre application

Comme vous l'avez compris, votre application devra être construite selon le **modèle d'architecture MVC**, comportant donc

- Une classe (ou des classes) représentant le **modèle (M)** de votre application : elle contiendra toutes les **données** de votre application et assurera à terme leur **sérialisation dans des fichiers** → cette classe (ou ces classes) devra **implémenter un interface** qui sera manipulé par le contrôleur → cette classe peut-être implémentée comme un singleton si vous le souhaitez → cette classe représente également la **« DAL (Data Access Layer) »** de votre application. Le modèle peut également être composé de plusieurs classes de type **DAO (Data Access Object)** où chaque **DAO** s'occupe de chaque entité de l'application.
- Une classe représentant la **vue (V)** de votre application : cette classe peut être la JFrame principale de votre application comprenant des méthodes permettant à l'utilisateur d'interagir avec elle, ou comporter un ensemble de méthodes permettant de faire apparaître des boites de dialogue afin d'interagir avec l'utilisateur → cette classe devra **implémenter un interface** qui sera manipulé par le contrôleur
- Une classe représentant le **contrôleur (C)** de votre application : cette classe dictera les **fonctionnalités et scénarios** possibles de l'application → cette classe possède des références vers le modèle (qui peut donc être composé de plusieurs classes de type DAO) et la vue, et n'implémente aucun interface autre que ceux nécessaires à la **gestion des événements** (comme ActionListener par exemple).

Cette manière de faire met bien en évidence les **principes SOLID**. Lors de l'évaluation finale, vous devrez être en mesure d'expliquer pourquoi.

Afin de vous aider au développement, un exemple complet est fourni ici : https://github.com/hepl-mesProjets/MVC_SOLID

2.5 Design Patterns et tests unitaires

Tout d'abord, comme le savez, l'entrée en session est une contrainte incontournable. Pour l'implémentation de cette fonctionnalité, on vous demande de respecter le patron de conception **« Template Method »** abordé au cours de Mr. Madani.

Tout au long du développement de votre application, on vous demandera de mettre en place les tests unitaires (utilisation de **JUnit**) pour tester :

- Vos **entités** (setters, getters, …)
- Les classes permettant **l'entrée en session** (voir « template method »)
- Les **classes DAO** de vos entités : les opérations CRUD (create, read, update, delete) pourront être testées

2.6 Librairie externe et création de jar : déploiement du délivrable

A terme,

- Votre application devra utiliser une **librairie externe** fournie sous forme d'un fichier jar ou mieux, par modification du fichier **pom.xml** de votre projet Maven.
- Vous devrez être capable de **créer** (**via IntelliJ** – par mise à jour du fichier **pom.xml**) un **fichier jar exécutable** de l'application contenant tous les fichiers nécessaires et manipulant la librairie externe. Le délivrable (le jar exécutable de l'application) sera alors testé sous Windows et sous Linux.

3. Méthodologie et étapes

Evaluation 1
Semaine du 14 avril 2025 en fonction de votre horaire de laboratoire

Porte sur :

- Analyse du projet (choix de la thématique et des fonctionnalités)
- Construction des packages, classes et interfaces métiers de l'application
- Création des classes de login selon le template method
- Mises en place des tests unitaires pour les entités et les classes de login
- Construction du package GUI et des classes de l'interface graphique de l'application (SANS AUCUNE GESTION d'EVENEMENT PRINCIPAUX) → il s'agit juste de la « coquille vide »

Etape 1 : Installation de IntelliJ IDEA, du JDK et choix de la thématique

Installez le JDK et IntelliJ en suivant les recommandations citées plus haut.

Identifiez la thématique sur laquelle vous allez travailler. Déterminez les besoins d'utilisateur auxquels vous allez répondre à travers des fonctionnalités. Une fois que vous savez ce que vous allez faire, réfléchissez aux classes « entités » dont vous avez besoin ainsi qu'aux relations entre elles, définissez les variables membres.

Avant de vous lancer dans le développement, vous devez faire valider votre projet par votre professeur de laboratoire avec qui vous devez avoir des échanges réguliers.

Etape 2 : Développement des classes métiers (« entités ») de l'application

Dans cette étape, vous allez créer les **classes métiers** (les **« entités »**) de votre application (**cas de l'application « Horaire »** : les classes Schedulable, Professor, Group, …). Pour cela, on vous demande

- De créer un ou plusieurs packages dans lesquels vous rangerez vos classes et interfaces par thème. Ou mieux, dans un package « entity » (voir capture du projet ci-dessus).
- Toute classe métier devra disposer de constructeurs, des accesseurs habituels, d'une méthode **toString()** et de la surcharge de la méthode **equals()**
- A chaque classe « entité » instanciable XXX devra correspondre une classe XXXtest situé dans le package miroir (situé dans le répertoire test de votre projet) permettant de réaliser des **tests unitaires** (utilisation de JUnit) sur les méthodes de la classe XXX.

Etape 3 : Développement des classes de login

Dans cette étape, vous allez créer les classes nécessaires à **l'entrée en session** de votre application, en respectant le **patron de conception « Template Method »**. Pour cela, on vous renvoie aux notes de cours de Mr. Madani sur les « Design Pattern ».

Dans cette étape, les logins et les mots de passe seront **stockés en mémoire dans un conteneur** (de type Map ou autre ; voir notes de Mr. Madani). Les classes correspondantes seront stockées dans un package « authentication » de votre projet (voir capture du projet ci-dessus).

On vous demande également de créer une classe de test qui permettra de réaliser les tests unitaires de vos classes de login.

Etape 4 : Développement des classes graphiques de l'application

En parallèle, vous devez créer l'interface graphique de base de votre application. On vous demande

- De créer un package GUI (ou View – voir capture ci-dessus) qui contiendra toutes les classes graphiques de votre application.
- De créer la JFrame principale de votre application.
- De créer les boites de dialogues (JDialog) modales personnalisées de votre application : chacune de ces classes disposera d'une méthode main de test permettant de faire apparaître la boite de dialogue, saisir les données et récupérer ces données dans le main de test (voir https://github.com/hepl-mesProjets/MVC_SOLID pour des exemples) → pas de tests unitaires avec JUnit ici !

Il ne s'agit pas ici de gérer tous les événements de votre fenêtre principale. Vous devez voir cette étape comme la **conception de la « maquette »** de votre application. Aucun lien ne doit encore être fait entre vos classes graphiques et vos classes métiers, ce qui permet un découpage net entre la **couche graphique de l'application** (« **GUI** » pour Graphic User Interface » - **la vue**) et la **couche des données** (communément appelée « **DAL** » pour Data Access Layer » - **le modèle**).

Evaluation 2
Jour d'examen de laboratoire en fonction de votre horaire d'examens

Porte sur :

- Gestion des événements – lien entre couche DAL (le **modèle**), GUI (la **vue**) et le **contrôleur** de votre application → mise en œuvre des principes SOLID
- Persistance des données et gestion des fichiers sérialisés, textes et properties + MAJ de l'entrée en session via le « Template Method »
- Création et utilisation des fichiers jar

Etape 5 : Gestion des événements de l'application → création du contrôleur

Il s'agit à présent de gérer les événements principaux de la fenêtre principale. Lors d'un clic sur un bouton, sur un item de menu, … votre application doit réagir en affichant une boîte de dialogue, mettre à jour un comboBox, etc…

Comme déjà mentionné, c'est le **contrôleur** qui agit en tant que chef d'orchestre, il doit

- implémenter les listeners correspondant aux événements principaux de votre interface graphique
- manipuler les données de votre application par l'intermédiaire du modèle
- manipuler l'interface graphique

Etape 6 : Persistance des données et fichiers (DAO et tests unitaires)

Il s'agit évidemment ici de créer/manipuler

- Le(s) fichiers properties à l'aide de la classe **Properties** → les logins et les mots de passe sont à présents stockées dans un fichier properties **users.properties** → on vous demande donc de créer la classe nécessaire à l'entrée en session en passant par ce fichier properties (utilisation du patron « template method » toujours)
- Le(s) fichiers sérialisés en utilisant les classes **ObjectOuputSteam** et **ObjectInputStream** → toutes vos classes « entités » doivent être stockées dans des conteneurs Java, ceux-ci devant être encapsulée dans des **classes DAO** (une classe par entité ou une seule classe DAL qui encapsule tous les conteneurs) → ces DAO (ou DAL) auront alors des méthodes **save()** et **load()** permettant de sérialiser/désérialiser dans des fichiers binaires tous vos objets « entités ».
- Les(s) fichiers texte ou csv en utilisant les classes **FileReader** et **BufferedReader** → en fonction des fonctionnalités de votre application.

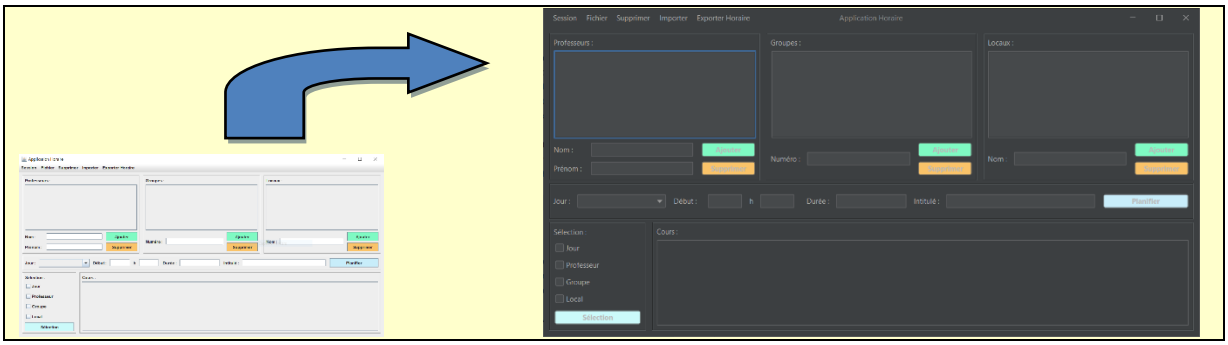
Toutes vos classes **DAO** devront être testées via **tests unitaires** (JUnit) → tester les différents opérations CRUD (create – read – update – delete).

En toute généralité, il s'agit de mettre à jour le modèle de votre application → cela ne devrait modifier en rien le contrôleur et la vue → néanmoins, à voir en fonction de votre choix d'application.

Etape 7 : Utilisation de libraire externe et création d'un jar exécutable

On vous demande ici :

- D'utiliser dans votre projet une **librairie externe** et d'en utiliser une des fonctionnalités. Si vous ne trouvez rien qui vous intéresse, vous utiliserez FlatLaf (« Flat Look And Feel »). Celui-ci vous permettra de modifier le « Look and Feel » de votre application. Celui-ci contient le thème utilisé par IntelliJ. Plus d'info sur : https://www.formdev.com/flatlaf/



- On vous demande de **créer le jar exécutable de votre application**. Le jar obtenu sera alors transféré sur un autre OS (Si développement sur Windows → Linux et, si développement sur Linux → Windows).

Dans les deux cas, vous allez devoir modifier le fichier **pom.xml** de votre projet, en ajoutant des **dépendances** au projet (pour l'utilisation d'une libraire existante) mais également d'utiliser le **Plugin Maven Shade** pour créer un jar exécutable autonome de votre application. Pour cela, on vous demande d'utiliser ChatGPT afin de vous guider 😉 !

Bon travail 😊 !