Projet Java POO: Système de Gestion de Bibliothèque

Développement d'une application avec interface graphique JavaFX et base de données

Vue d'ensemble du projet

Développer une application complète de gestion de bibliothèque avec interface graphique JavaFX et base de données.

Projet Java : Système de Gestion de Bibliothèque
Vue d'ensemble

Objectif : Développer une application de gestion de bibliothèque avec interface graphique JavaFX et base de données.

Description du projet
Les étudiants développeront une application complète permettant de gérer une bibliothèque : livres, membres, emprunts et retours. L'application intègrera tous les concepts Java essentiels dans un contexte pratique et professionnel.

Fonctionnalités à implémenter

1. Gestion des livres
- Ajouter, modifier, supprimer des livres
- Rechercher des livres par titre, auteur ou ISBN
- Afficher la liste complète des livres

2. Gestion des membres
- Enregistrer de nouveaux membres
- Modifier les informations des membres
- Visualiser l'historique des emprunts

3. Gestion des emprunts
- Emprunter un livre
- Retourner un livre
- Calculer les pénalités de retard
- Afficher les emprunts en cours

Objectifs du projet

Gestion complète

Application pour gérer **livres**, **membres**, **emprunts** et **retours**

POO & Architecture

Application des concepts de **programmation orientée objet** et mise en œuvre d'une **architecture en couches**

Interface graphique

Développement d'une **interface utilisateur** intuitive avec **JavaFX**

Base de données

Intégration d'une **base de données** pour la persistance des informations

Fonctionnalités de gestion des livres

Ajouter

Créer de **nouveaux livres** dans le système avec toutes les informations nécessaires

Modifier

Mettre à jour les **informations** d'un livre existant (titre, auteur, etc.)

Supprimer

Rechercher

Fonctionnalités de gestion des membres

Enregistrer

Créer de **nouveaux membres** dans le système avec informations complètes

Modifier

Mettre à jour les **coordonnées** et informations d'un membre existant

Fonctionnalités de gestion des emprunts

Emprunter un livre

Enregistrer un **nouvel emprunt** avec date de début et date de retour prévue

Retourner un livre

Marquer un livre comme **retourné** et mettre à jour sa disponibilité

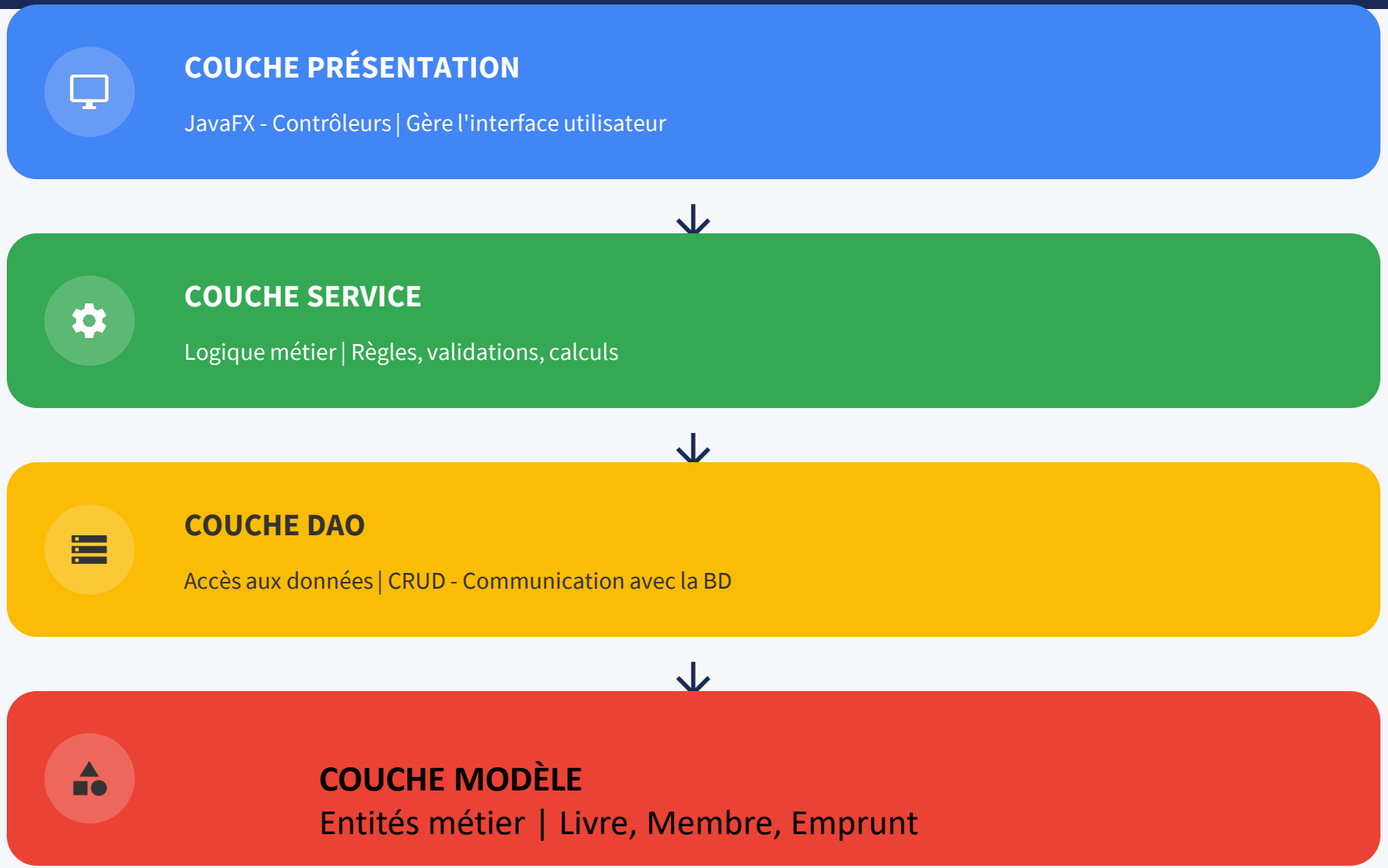
Calculer les pénalités

Déterminer automatiquement les **frais de retard** selon le nombre de jours

Afficher les emprunts en cours

Visualiser tous les **emprunts actifs** avec leurs dates d'échéance

Architecture en couches



Règles importantes de l'architecture

Communication entre couches

Le Contrôleur **NE PARLE JAMAIS** directement au DAO

Rôle du Service

Le Service est le **chef d'orchestre** qui:

- Coordonne **plusieurs DAO**
- Applique les **règles métier**
- Gère les **transactions**

Rôle du DAO

Le DAO ne fait **QUE** des opérations de base de données:

- **CRUD** (Create, Read, Update, Delete)
- **PAS** de logique métier
- **PAS** de calculs complexes

Étape 1: Le Modèle (Les Données)

Concepts POO à appliquer

- **Classes & Attributs**
- **Encapsulation**
- **Héritage**
- **Polymorphisme**

Ordre de création

1. **Document** (abstraite)
2. **Empruntable** (interface)
3. **Livre**
4. **Magazine**
5. **Membre**
   Emprunt

Tests

Créer TestModele.java pour vérifier le fonctionnement avant de continuer

Structure des classes du modèle

Document
- id: String
- titre: String
- **calculerPenaliteRetard()** (abstraite)

Empruntable
- peutEtreEmprunte()
- emprunter()
- retourner()

Livre
- auteur: String
- anneePublication: int
- **extends Document**
- **implements Empruntable**

Magazine
- numero: int
- mois: String
- **extends Document**
- **implements Empruntable**

Personne
- nom: String
- prenom: String
- email: String

Membre
- id: int
- actif: boolean
- **extends Personne**

Emprunt
- dateEmprunt: Date
- dateRetourPrevue: Date
- livre: Livre
- membre: Membre

Étape 2: Les Exceptions

Exceptions personnalisées

ValidationException

Erreurs de saisie (email invalide, ISBN mal formé)

LivreIndisponibleException

Livre déjà emprunté ou inexistant

MembreInactifException

Membre désactivé ne peut pas emprunter

LimiteEmpruntDepasseeException

Membre a déjà 3 emprunts en cours

throws

Déclare qu'une méthode **peut lever** une exception
Utilisé dans la signature de méthode
Transmet la responsabilité de gestion

throw

Lève explicitement une exception
Utilisé dans le corps de méthode
Crée une nouvelle instance d'exception

- **Messages d'erreur clairs**
- **Traitement spécifique**
- **Code plus lisible**
- **Séparation des responsabilités**

Étape 2: Les Exceptions

Exceptions personnalisées

Approche avec exceptions personnalisées

❌ Mauvaise approche (avec String)

```
public Livre emprunter(String isbn) {
        Livre livre = trouverLivre(isbn);
        if (livre == null) {
                return null;
                // ❌ On ne sait pas POURQUOI ça a échoué
        }
        return livre;
}
```

✓ Bonne approche (avec exceptions)

```
public Livre emprunter(String isbn) throws LivreIndisponibleException {
        Livre livre = trouverLivre(isbn);
        if (livre == null || !livre.isDisponible()) {
                throw new LivreIndisponibleException(isbn);
                // ✅ Message clair + ISBN concerné
        }
        return livre;
}
```

✓ Gestion avec try-catch

```
try {
        Livre livre = service.emprunterLivre("978-123456");
        showSuccess("Livre emprunté");
}
catch (LivreIndisponibleException e) {
        showError("Livre indisponible", e.getMessage());
}
catch (MembreInactifException e) {
        showError("Membre inactif", e.getMessage());
}
```

Étape 3: Base de données et Singleton

Création de la base de données

livres	ISBN, titre, auteur...
membres	id, nom, email...
emprunts	clés étrangères vers livres et membres

Avantages du Singleton

- **Performance**
- **Ressources**
- **Cohérence**

Problème

Plusieurs threads peuvent créer simultanément plusieurs instances du Singleton

Solution: Double-Checked Locking

Vérification avant et après le verrou pour garantir une seule instance

Étape 3: Base de données et Singleton

Pattern Singleton

```
public class DatabaseConnection {
	// 1. Instance unique statique
	private static DatabaseConnection instance;
	// 2. Connexion à partager private Connection connection;
	// 3. Constructeur PRIVÉ
	private DatabaseConnection() {
	// Créer la connexion }
	// 4. Méthode publique pour obtenir l'instance
	public static DatabaseConnection getInstance() {
		if (instance == null) {
				instance = new DatabaseConnection();
		}
		return instance;
	}
}
```

```
 // Double-Checked Locking
public static DatabaseConnection getInstance() {
		if (instance == null) {
				// 1er check (rapide)
				synchronized (DatabaseConnection.class) {
						if (instance == null) {
								// 2ème check
								instance = new
DatabaseConnection();
						}
				}
		}
		return instance;
}
```

```
 // Utilisation dans les DAO
public class LivreDAOImpl implements LivreDAO {
		private Connection connection;
		public LivreDAOImpl() {
				// Tous les DAO partagent LA MÊME connexion
				this.connection =
DatabaseConnection.getInstance().getConnection();
		}
}
```

Étape 4: Pattern DAO

Architecture DAO

DAO<T> (interface)

Interface générique avec méthodes CRUD de base

LivreDAO (interface)

Méthodes spécifiques: findByAuteur(), findByTitre()

LivreDAOImpl

Implémentation concrète avec code SQL

CRUD complet

- **Create (save)**
- **Read (findById)**
- **Read (findAll)**
- **Update**
- **Delete**

Méthode utilitaire: mapResultSetToLivre()

Évite la duplication de code

Utilisée par toutes les méthodes de lecture

```
private Livre mapResultSetToLivre(ResultSet rs) throws
SQLException { Livre livre = new Livre();
livre.setIsbn(rs.getString("isbn"));
livre.setTitre(rs.getString("titre"));
livre.setAuteur(rs.getString("auteur")); // ... return livre;
}
```

Étape 4: Pattern DAO

PreparedStatement vs Statement

DANGEREUX: Statement

```
String isbn = userInput; String sql = "SELECT * FROM livres WHERE isbn = '" + isbn + "'"; // Risque d'injection SQL!
```

SÉCURISÉ: PreparedStatement

```
String sql = "SELECT * FROM livres WHERE isbn = ?"; PreparedStatement stmt = conn.prepareStatement(sql); stmt.setString(1, isbn); // Échappement automatique!
```

```
public class LivreDAOImpl implements LivreDAO { private Connection connection; public LivreDAOImpl() { // Utilise le Singleton pour la connexion this.connection = DatabaseConnection.getInstance() .getConnection(); } @Override public void save(Livre livre) throws SQLException { // Utilise this.connection pour les requêtes try (PreparedStatement stmt = connection.prepareStatement(sql)) { // ... } } }
```

- **Connexion partagée**
- **Facile à tester**

Étape 5: La Couche Service

Rôle du Service

Le Service est le chef d'orchestre qui coordonne les opérations

Responsabilité	DAO	Service
Requêtes SQL (CRUD)	X	
Règles métier (calculs, conditions)		X
Validation (logique et cohérence)		X
Coordination des DAO		X
Gestion des Transactions		X
Mapping des données (ResultSet vers Objet)	X	

DAO vs Service

Recherche simple (DAO)

Recherche par titre, auteur ou ISBN sans logique complexe

Recherche avancée (Service)

Combine plusieurs recherches, élimine les doublons

Étape 6: Contrôleurs et JavaFX

Pattern MVC

MODÈLE

Livre.java
Membre.java
Emprunt.java

VUE

MainView.fxml
TableView
TextField

CONTRÔLEUR

MainController.java
@FXML
handleAjouter()

JavaFX: Composants clés

FXML

XML pour définir l'interface

TableView

Affichage des données en tableau

Événements

Gestion des interactions

@FXML

Injection des composants

Appeler le DAO depuis le Contrôleur

Le contrôleur ne doit pas contourner la couche Service

Oublier le @FXML

Les champs ne seront jamais liés aux éléments FXML

Nom différent entre fx:id et champ Java

L'injection ne fonctionnera pas

Getter manquant pour PropertyValueFactory

JavaFX ne pourra pas récupérer la valeur à afficher

Étape 7: Intégration finale

Assemblage des composants

1. **Créer la classe Main.java**
   Point d'entrée de l'application JavaFX

2. **Charger le fichier FXML principal**
   Définir l'interface utilisateur

3. **Configurer la scène et la fenêtre**
   Titre, dimensions et affichage

4. **Gérer la fermeture de l'application**
   Fermer la connexion à la base de données

Étape 7: Intégration finale

Checklist finale

Modèle

- Classes compilent
- Héritage fonctionne
- Interfaces implémentées
- Méthodes de calcul

Base de données

- Tables créées
- Données de test
- Clés étrangères
- Singleton fonctionnel

DAO & Service

- CRUD complet
- PreparedStatement
- Logique métier
- Gestion des exceptions

Interface JavaFX

- Liaison FXML ↔ Contrôleur
- TableView fonctionnelle
- Boutons et événements
- Messages d'erreur

Questions fréquentes des étudiants (Q1-Q2)

Q1 Pourquoi ne pas appeler directement le DAO depuis le Contrôleur?

✕ SANS Service

```
// Dans le Contrôleur @FXML private
void handleEmprunter() {
            // Le contrôleur doit faire
TOUTE       la logique
            Livre livre =
livreDAO.findById(isbn);
            if (livre == null) {

            showError("Livre
inexistant");
return;
}
if (!livre.isDisponible()) {
            showError("Livre
indisponible"); return;
} // ... encore 20 lignes de code
}
```

✓ AVEC Service

```
 // Dans le Contrôleur (simple et clair)
@FXML
private void handleEmprunter() {
try {
            Emprunt emprunt =
empruntService.emprunterLivre(isbn,
idMembre);
showSuccess("Emprunt enregistré");
} catch (LivreIndisponibleException e) {
showError("Livre indisponible");
}
}
```

Réutilisabilité

Code utilisable pour d'autres applications (mobile, web)

Testabilité

Possibilité de tester le Service sans JavaFX

Maintenance

Une modification = un seul endroit à changer

Clarté

Chaque couche a un rôle bien défini

Singleton

✗ SANS Singleton

Problèmes:

- MySQL limite à 151 connexions par défaut
- Ouvrir une connexion prend 100-200ms
- Consommation de mémoire inutile
- 10 utilisateurs = 30 connexions!

✓ AVEC Singleton

Avantages:

- **1 seule connexion** pour toute l'application
- Performance optimisée
- Gestion efficace des ressources
- Cohérence des transactions

Métaphore

Sans Singleton: Chaque employé achète sa propre imprimante

Avec Singleton: Tous les employés partagent 1 imprimante centrale

Conseils de méthodologie

Travail en équipe efficace

- Répartition par **couche**: Modèle, DAO, Service, JavaFX
- Répartition par **fonctionnalité**: Livres, Membres, Emprunts
- Définir les interfaces **ensemble** avant de se séparer
- Tester régulièrement l'**intégration**

Utilisation de Git

Structure des **branches**:

main	**dev**	**feature/***	**hotfix/***
Code stable	Développement	Fonctionnalités	Corrections

Workflow: Créer une branche → Travailler → Commit → Push → Merge

Répartition des Modules

LIVRES

Étudiant A

MEMBRES

Étudiant B

EMPRUNTS

Étudiant C

INFRASTRUCTURE

Étudiant D

Introduction à la Répartition par Modules

Approche par Fonctionnalités

Chaque étudiant gère un module **complet**

Du **modèle** à l'**interface** utilisateur

Responsabilité **de bout en bout**

Avantages

- **Travail en parallèle** efficace
- **Autonomie** maximale
- **Intégration** simplifiée

Module LIVRES

Gestion complète du catalogue de livres

Module MEMBRES

Gestion complète des adhérents

Module EMPRUNTS

Logique métier des emprunts

Infrastructure

Vue d'Ensemble de l'Architecture

Interface Utilisateur

JavaFX - FXML

LIVRES

Étudiant A

MEMBRES

Étudiant B

EMPRUNTS

Étudiant C

Infrastructure

Étudiant D

Base de Données

MySQL

Étudiant A - Module LIVRES

Fichiers à créer

model/

- Document.java (classe abstraite)
- Livre.java
- Empruntable.java (interface)

dao/

- LivreDAO.java (interface)
- impl/LivreDAOImpl.java

service/

- BibliothequeService.java (partie livres)

controller/

- LivreController.java

resources/fxml/

- LivreView.fxml

sql/

- CREATE TABLE livres + INSERT données test

Méthodes à implémenter

DAO

save | findById | findAll | update | delete | findByAuteur | findByTitre | findDisponibles

Service

ajouterLivre | modifierLivre | supprimerLivre | rechercherLivres | getLivresDisponibles

Controller

CRUD complet | recherche | statistiques

Livre.java doit étendre Document et implémenter Empruntable. Le contrôleur doit gérer l'affichage dans un TableView avec boutons d'action.

Étudiant B - Module MEMBRES

Fichiers à créer

model/

├── Personne.java (classe abstraite)

└── Membre.java

dao/

├── MembreDAO.java (interface)

└── impl/MembreDAOImpl.java

service/

└── BibliothequeService.java (partie membres)

controller/

└── MembreController.java

resources/fxml/

└── MembreView.fxml

sql/

└── CREATE TABLE membres + INSERT données test

Méthodes à implémenter

DAO

save findById findAll update delete findByEmail findActifs

Service

ajouterMembre modifierMembre activerDesactiver rechercherMembres getHistorique

Controller

CRUD complet activation/désactivation historique emprunts

Membre.java doit **étendre Personne**. Le contrôleur doit gérer l'**activation/désactivation** des membres et afficher leur **historique d'emprunts**.

مدرسة علوم المعلومات
ⵜⵉⵏⵎⵍ ⵏ ⵜⵓⵙⵙⵏⴰ ⵏ ⵉⵙⴰⵍⵍⵏ
ECOLE DES SCIENCES DE L'INFORMATION

Étudiant C - Module EMPRUNTS

Fichiers à créer

model/

└── Emprunt.java

dao/

├── EmpruntDAO.java (interface)

└── impl/EmpruntDAOImpl.java

service/

└── EmpruntService.java (TOUTE la logique métier)

controller/

└── EmpruntController.java

resources/fxml/

└── EmpruntView.fxml

sql/

└── CREATE TABLE emprunts + INSERT données test

Méthodes à implémenter

DAO

- save
- findById
- findAll
- update
- findByMembre
- findEnCours
- countEmpruntsEnCours

Service

- emprunterLivre
- retournerLivre
- getEmpruntsEnRetard
- calculerPenalite

Controller

- formulaire emprunt
- formulaire retour
- gestion exceptions

Le service doit implémenter la validation des 3 emprunts maximum par membre et le calcul des pénalités pour les retours en retard. Le contrôleur doit gérer toutes les exceptions métier.

Étudiant D - Infrastructure & Intégration

Fichiers à créer

util/
- ├— DatabaseConnection.java (Singleton)
- ├— StringValidator.java
- └— DateUtils.java

exception/
- ├— ValidationException.java
- ├— LivreIndisponibleException.java
- ├— MembreInactifException.java
- └— LimiteEmpruntDepasseeException.java

dao/
- └— DAO.java (interface générique)

controller/
- └— MainController.java (navigation)

resources/fxml/
- └— MainView.fxml (TabPane principal)

Main.java (point d'entrée)

sql/
- └— create_database.sql (script complet)

README.md

pom.xml

Tâches spécifiques

- **Singleton** avec connexion MySQL
- Validation **ISBN, email, dates**
- Toutes les **exceptions personnalisées**
- Navigation entre **onglets A/B/C**
- **Intégration finale** + tests
- Configuration **pom.xml**
- Documentation **README.md**
- Point d'entrée **Main.java**

Le MainController doit gérer la navigation entre les onglets et coordonner l'interaction entre les différents modules. Le Singleton DatabaseConnection doit garantir une connexion unique à la base de données.

Livrables Attendus

Étudiant A

Module LIVRES

- Code complet du module (modèle, DAO, service, contrôleur)
- Tests unitaires pour toutes les classes
- Documentation JavaDoc des méthodes publiques
- Script SQL de création de table

Étudiant B

Module MEMBRES

- Code complet du module (modèle, DAO, service, contrôleur)
- Tests unitaires pour toutes les classes
- Documentation JavaDoc des méthodes publiques
- Script SQL de création de table

Étudiant C

Module EMPRUNTS

- Code complet du module (modèle, DAO, service, contrôleur)
- Tests unitaires pour toutes les classes
- Documentation JavaDoc des méthodes publiques
- Script SQL de création de table

Étudiant D

Infrastructure

- Singleton DatabaseConnection
- Utilitaires de validation
- Toutes les exceptions personnalisées
- MainController + intégration finale

Livrables Attendus

Livrables Collectifs: Pour le groupe

- Application fonctionnelle
- Rapport de projet
- Présentation
- Dépôt Git complet

Critères de qualité

- Code fonctionnel sans bug
- Respect conventions Java
- Tests > 80% de couverture
- Documentation complète
- Intégration fluide
- Respect des délais