PROJET DE PROGRAMMATION ORIENTÉE OBJET II

Système de gestion d'événements distribués

Réalisé par : DJUITCHOKO BRAYANN

Martricule : 22P207

Encadrant : **Dr. Kungne**

ENSPY – Département du Genie Informatique

Année universitaire 2024–2025

Table des matières

Introduction

L'objectif principal de ce projet, réalisé dans le cadre du cours de Programmation Orientée Objet, était de mettre en application les concepts avancés de la POO. Il s'agissait de concevoir et d'implémenter un système de gestion d'événements (tels que des conférences et des concerts) fonctionnel et robuste. Ce système devait permettre la gestion des événements eux-mêmes, l'inscription des participants, la distinction des organisateurs, la mise en place de notifications en temps réel lors de modifications ou d'annulations, et la persistance des données.

Au-delà des fonctionnalités de base, le projet visait à explorer l'utilisation de design patterns pertinents (Observer, Singleton), la création et la gestion d'exceptions personnalisées, la manipulation efficace de collections avec les apports de Java moderne (Streams, Lambdas), ainsi que la sérialisation des données pour assurer leur sauvegarde et leur chargement. Une extension significative a été l'ajout d'une interface graphique utilisateur (GUI) pour rendre le système interactif et exploitable de manière conviviale.

Ce rapport détaille les choix de conception architecturale du backend et de l'interface graphique, les défis techniques rencontrés, les solutions apportées, ainsi que la stratégie de tests mise en œuvre pour valider le bon fonctionnement du système.

1 Choix de Conception et Architecture Backend

La fondation du système repose sur une architecture backend solide, pensée pour être évolutive et maintenable, en respectant les principes de la Programmation Orientée Objet.

1.1 Modélisation des Classes (Structure Générale)

Le cœur du système est modélisé autour de plusieurs entités clés :

- **Evenement (Classe Abstraite)** : Sert de classe de base pour tous les types d'événements. Elle factorise les attributs communs (`ID`, `nom`, `date et heure`, `lieu`, `capacité maximale`) et les comportements essentiels (`ajouterParticipant()`, `annuler()`, `afficherDetails()`). L'utilisation de l'abstraction ici permet de manipuler différents types d'événements de manière polymorphique (par exemple, dans une liste d'événements). Elle étend également `EvenementObservable` pour intégrer le mécanisme de notification.
- **Conference et Concert (Classes Concrètes)** : Ces classes héritent de `Evenement` et ajoutent des attributs spécifiques. `Conference` inclut un *thème* et une liste d'intervenants (`List<Intervenant>`), tandis que `Concert` possède un *artiste principal* et un *genre musical*. Cela démontre l'application de l'héritage pour spécialiser des comportements et des données.
- **Intervenant (Classe Simple)** : Une classe de données pour représenter les intervenants d'une conférence, avec leur `nom` et `spécialité`.
- **Participant (Classe Concrète)** : Représente un utilisateur s'inscrivant à un événement. Chaque participant possède un `ID`, un `nom` et un `email`. Cette classe implémente l'interface `ParticipantObserver`, lui permettant de recevoir des notifications des événements auxquels il est inscrit.
- **Organisateur (Classe Concrète)** : Hérite de `Participant`, ajoutant la capacité de gérer une liste d'événements qu'il a créés ou qu'il supervise (`List<Evenement> evenementsOrganises`).

L'héritage a été utilisé pour créer une hiérarchie claire et logique, favorisant la réutilisation du code et le polymorphisme. Par exemple, `GestionEvenements` peut stocker et manipuler des `Conference` et des `Concert` de manière transparente à travers la référence `Evenement`.

1.2 Services et Interfaces

Pour découpler les composants et favoriser une architecture flexible :

— **`NotificationService` (Interface)** : Définit un contrat pour l'envoi de notifications avec une unique méthode `envoyerNotification(String message)`. Ce choix permet d'abstraire le mécanisme de notification.
— **`EmailNotificationService` (Implémentation Concrète)** : Une implémentation de `NotificationService` qui, dans ce projet, simule l'envoi d'une notification par email (affichage console). L'important est que le reste du système (par exemple, la classe `Participant`) dépend de l'interface `NotificationService`, et non de cette implémentation spécifique, respectant ainsi le principe d'inversion des dépendances.

1.3 Design Patterns Utilisés

Plusieurs design patterns ont été intégrés pour résoudre des problèmes de conception courants :

— **Singleton (`GestionEvenements`)** :
  — **Description** : La classe `GestionEvenements` est implémentée en tant que Singleton. Elle assure qu'une seule instance de cette classe existe dans toute l'application, fournissant un point d'accès global et centralisé pour la gestion de tous les événements (ajout, suppression, recherche).
  — **Justification** : Ce pattern a été choisi pour centraliser la logique de gestion des événements et éviter les incohérences de données qui pourraient survenir si plusieurs instances de gestionnaires d'événements coexistaient. Il simplifie l'accès à la collection principale d'événements depuis différentes parties de l'application, notamment les contrôleurs de l'interface graphique.
— **Observer (`EvenementObservable`, `ParticipantObserver`)** :
  — **Description** : Ce pattern est au cœur du système de notifications en temps réel. La classe `Evenement` (via sa super-classe `EvenementObservable`) joue le rôle du "Sujet" (Observable). Elle maintient une liste d'observateurs et notifie ces derniers lors de changements d'état (modification, annulation). La classe `Participant` implémente l'interface `ParticipantObserver` et joue le rôle de l'"Observateur".
  — **Justification** : L'Observer a été choisi pour découpler les événements des participants. Un événement n'a pas besoin de connaître les détails spécifiques de la manière dont chaque participant souhaite être notifié. Lorsqu'un événement

est modifié (par exemple, sa date, son lieu) ou annulé, tous les participants inscrits (qui sont des observateurs) sont automatiquement informés via leur méthode `recevoirNotification()`. Cela rend le système flexible et facile à étendre (par exemple, ajouter d'autres types d'observateurs).

— **(Potentiel) Factory Pattern** : Bien que non explicitement implémenté pour la création d'événements dans la version finale (la création étant gérée directement via les constructeurs dans le dialogue de l'interface graphique), une `EventFactory` aurait pu être envisagée si le processus de création d'objets `Evenement` était devenu plus complexe, par exemple, en fonction de données externes ou de multiples paramètres de configuration.

1.4 Respect des Principes SOLID

Un effort a été fait pour adhérer aux principes SOLID afin de garantir une conception de haute qualité :

— **S (Single Responsibility Principle - Principe de Responsabilité Unique)** :
  — `GestionEvenements` est responsable de la gestion de la collection d'événements.
  — `Evenement` et ses sous-classes gèrent les informations et la logique d'un seul événement.
  — `NotificationService` est uniquement responsable de l'envoi de notifications.
  — `SerialisationUtil` s'occupe de la persistance des données.
  — Les contrôleurs FXML (ex : `MainViewController`, `EventEditDialogController`) gèrent la logique de leur vue respective.
— **O (Open/Closed Principle - Principe Ouvert/Fermé)** :
  — Le système est ouvert à l'extension : on peut ajouter de nouveaux types d'événements (ex : `Atelier`) en héritant d'`Evenement` sans modifier le code existant de `GestionEvenements` ou des mécanismes de notification. De même, de nouvelles implémentations de `NotificationService` pourraient être ajoutées.
  — Il est fermé à la modification : les classes existantes n'ont (idéalement) pas besoin d'être modifiées pour supporter ces extensions.
— **L (Liskov Substitution Principle - Principe de Substitution de Liskov)** :
  — Les instances de `Conference` et `Concert` peuvent être utilisées partout où une instance d'`Evenement` est attendue (par exemple, dans `List<Evenement>`, `Map<String, Evenement>` de `GestionEvenements`, ou comme argu-

ment de méthode attendant un `Evenement`).

- **I (Interface Segregation Principle - Principe de Ségrégation des Interfaces)** :
  - `NotificationService` est une interface concise et ciblée sur un seul besoin.
  - `ParticipantObserver` est également une interface spécifique pour le rôle d'observateur, ne demandant que l'implémentation de `recevoirNotification()`.
- **D (Dependency Inversion Principle - Principe d'Inversion des Dépendances)** :
  - La classe `Participant` dépend de l'interface `NotificationService`, et non d'une implémentation concrète comme `EmailNotificationService`. Cela permet d'injecter différentes stratégies de notification si nécessaire.
  - Le pattern Observer contribue également à ce principe : les sujets (`EvenementObservable` ne dépendent pas des observateurs concrets (`Participant`), mais de l'abstraction `ParticipantObserver`.

1.5 Gestion des Exceptions Personnalisées

Pour une gestion des erreurs plus claire et spécifique, des exceptions personnalisées ont été créées :

- **`CapaciteMaxAtteinteException`** : Levée lorsqu'on tente d'inscrire un participant à un événement qui a déjà atteint sa capacité maximale.
- **`EvenementDejaExistantException`** : Levée lors de la tentative d'ajout d'un événement avec un ID qui existe déjà dans `GestionEvenements`.
- **`ParticipantDejaInscritException`** : Levée si un participant tente de s'inscrire plusieurs fois au même événement.
- **`EvenementNonTrouveException`** : Levée si une opération est tentée sur un événement qui n'est pas trouvé dans le système (par exemple, lors d'une suppression ou d'une recherche par ID infructueuse).

L'utilisation de ces exceptions personnalisées rend le code plus expressif et permet aux appelants (notamment l'interface graphique) de réagir de manière plus appropriée aux différentes situations d'erreur.

1.6 Manipulation de Collections et Fonctionnalités Java Modernes

Le projet tire parti des fonctionnalités modernes de Java pour la manipulation des données :

- **Collections Génériques** : Utilisation intensive de `List` (par exemple, `participantsInscrit` dans `Evenement`, `listeIntervenants` dans `Conference`) et `Map` (par exemple,

`evenements` dans `GestionEvenements`) pour stocker et gérer les objets de manière typée et sécurisée.

— **Streams et Lambdas** : Employés pour des opérations sur les collections, rendant le code plus concis et lisible. Par exemple :
  — Pour notifier les observateurs dans `EvenementObservable` : `new ArrayList<>(obser -> observateur.recevoirNotification(message));`
  — Pour les méthodes de recherche avancée dans `GestionEvenements` (recherche par nom, par date).
  — Pour transformer des listes d'objets en chaînes de caractères pour l'affichage (par exemple, la liste des intervenants).

— **`Optional`** : Utilisé pour les retours de méthodes de recherche comme `GestionEvenements.idEvent)`, afin de gérer de manière plus élégante les cas où un événement pourrait ne pas être trouvé, évitant ainsi les `NullPointerException` directes.

— **`LocalDateTime`** : La classe `java.time.LocalDateTime` a été utilisée pour représenter la date et l'heure des événements, offrant une API moderne et robuste pour la manipulation des dates et heures.

1.7 Sérialisation/Désérialisation des Données

La persistance des données est gérée par sérialisation/désérialisation en JSON.

— **Choix de Jackson** : La bibliothèque Jackson a été choisie pour sa popularité, sa flexibilité et ses performances pour la manipulation de JSON en Java.

— **Configuration pour le Polymorphisme** : Un défi majeur était de gérer la sérialisation et la désérialisation correctes de la hiérarchie d'objets `Evenement` (avec ses sous-classes `Conference` et `Concert`). Cela a été résolu en utilisant les annotations Jackson :
  — `@JsonTypeInfo` sur la classe `Evenement` pour indiquer comment l'information de type doit être incluse dans le JSON.
  — `@JsonSubTypes` sur `Evenement` pour lister les sous-classes concrètes.
  — `@JsonIgnoreProperties({"observateurs"})` sur `Evenement` pour éviter de sérialiser la liste des observateurs, qui n'est pas destinée à être persistée et pourrait causer des problèmes.
  — Des constructeurs par défaut ont été ajoutés aux classes du modèle pour faciliter la désérialisation par Jackson.

— **Classe `SerialisationUtil`** : Une classe utilitaire a été créée pour encapsuler la logique de sauvegarde (`sauvegarderEvenementsJSON`) et de chargement (`chargerEvenementsJSON`) des listes d'événements. Elle configure

l'`ObjectMapper` de Jackson, notamment avec `JavaTimeModule` pour la gestion de `LocalDateTime`.

— **Format de Données** : Les données sont sauvegardées dans un fichier `evenements_data.jso`

1.8 Programmation Événementielle et Asynchrone

— **Programmation Événementielle** : Le pattern Observer est la principale manifestation de la programmation événementielle dans ce projet. Les changements d'état dans un objet `Evenement` (sujet) déclenchent des actions (notifications) dans d'autres objets (`Participant`, observateurs).

— **Programmation Asynchrone (Bonus)** : Pour illustrer ce concept, une méthode `envoyerNotificationAsync(String message, long delaiMillis)` a été ajoutée à `EmailNotificationService`. Elle utilise `CompletableFuture.runAsync` pour simuler l'envoi d'une notification sans bloquer le thread principal, ce qui serait utile pour des opérations potentiellement longues dans une application réelle.

2 Conception et Implémentation de l'Interface Graphique (GUI)

Pour rendre le système utilisable et démontrer ses fonctionnalités, une interface graphique a été développée.

2.1 Choix Technologiques

— **JavaFX** : Choisi comme framework pour l'interface graphique en raison de sa modernité par rapport à Swing, de ses capacités de stylisation via CSS, et de son intégration avec FXML.

— **FXML** : Utilisé pour définir la structure et l'agencement des vues de l'interface utilisateur. Cela permet une séparation claire entre la conception de l'interface (le "quoi") et la logique applicative (le "comment"), qui réside dans les classes de contrôleurs Java.

— **CSS** : Un fichier `styles.css` de base a été utilisé pour améliorer l'apparence visuelle des composants JavaFX.

— **Scene Builder (outil externe)** : Scene Builder est l'outil qui a été utilisé pour concevoir et modifier visuellement de tels fichiers, facilitant grandement le développement d'interfaces.

2.2 Architecture de l'Interface

L'interface graphique suit une approche inspirée du modèle MVC (Modèle-Vue-Contrôleur) :

— **Modèle (Model)** : Constitué par les classes du backend (`Evenement`, `GestionEvenements`, etc.).
— **Vue (View)** : Définie par les fichiers FXML (`MainView.fxml` pour la fenêtre principale, `EventEditDialog.fxml` pour le dialogue de création/édition d'événements).
— **Contrôleur (Controller)** : Les classes Java associées aux fichiers FXML (`MainViewController`, `EventEditDialogController`). Elles contiennent la logique pour répondre aux interactions de l'utilisateur, mettre à jour la vue, et interagir avec le modèle (via `GestionEvenements`).

La fenêtre principale (`MainView.fxml`) est structurée avec un `BorderPane` contenant une barre de menus, une barre d'outils, une `TableView` pour lister les événements, et un panneau de détails. Le dialogue d'édition (`EventEditDialog.fxml`) utilise un `GridPane` pour organiser les champs de saisie.

2.3 Fonctionnalités Clés de l'Interface

L'interface graphique permet de :

— **Visualiser les événements** : Une `TableView` affiche la liste des événements avec leurs informations principales (ID, nom, date, lieu, type, capacité, nombre d'inscrits).
— **Afficher les détails d'un événement** : La sélection d'un événement dans la table met à jour un panneau de détails affichant toutes ses informations, y compris les champs spécifiques aux conférences (thème, intervenants) et aux concerts (artiste, genre).
— **Créer un nouvel événement** : Un dialogue modal permet de saisir les informations d'un nouvel événement (type, ID, nom, date, etc.). Des champs conditionnels s'affichent en fonction du type d'événement choisi (Conférence ou Concert).
— **Modifier un événement existant** : Le même dialogue, pré-rempli avec les données de l'événement sélectionné, permet de modifier ses propriétés.
— **Supprimer un événement** : Après confirmation, l'événement sélectionné est supprimé du système (ce qui déclenche également la notification d'annulation aux participants inscrits).
— **S'inscrire/Se désinscrire (simulé)** : Un bouton permet à un utilisateur simulé (`currentUser` dans `MainViewController`) de s'inscrire à un événement. Le

bouton est désactivé si l'événement est complet ou si l'utilisateur est déjà inscrit. La désinscription n'a pas été explicitement implémentée via un bouton dédié dans cette version, mais le mécanisme existe dans le backend.

— **Rechercher et Filtrer les événements** : Un champ de recherche permet de filtrer dynamiquement la liste des événements affichés dans la table en fonction du nom, du lieu, de l'ID ou du type.

— **Notifications UI** : Une zone de notification en bas de la fenêtre principale affiche des messages à l'utilisateur (par exemple, confirmation de sauvegarde, résultat d'une inscription). Les notifications du pattern Observer (par exemple, si un événement est modifié par une action et que `currentUser` y est inscrit) sont également relayées à cette zone.

— **Sauvegarder et Charger les données** : Des options de menu permettent de sauvegarder l'état actuel des événements dans le fichier JSON et de recharger les données depuis ce fichier.

2.4 Défis Rencontrés lors du Développement de la GUI

Le développement de l'interface graphique a présenté plusieurs défis techniques, notamment liés à l'intégration de JavaFX dans un projet Maven utilisant le système de modules de Java (JPMS) :

— **Configuration du Système de Modules (JPMS)** :

  — `IllegalAccessException` : Initialement, JavaFX ne pouvait pas accéder à la classe `MainGuiApp` car le package `com.gestionevents.gui` n'était pas exporté. La solution a été de créer un fichier `module-info.java` et d'y ajouter `exports com.gestionevents.gui;`.

  — `InvalidModuleDescriptorException (Package not found)` : Une erreur de nom de package dans `module-info.java` (référence à `com.cours.poo.gest` au lieu de `com.gestionevents.model`) a dû être corrigée.

  — Il a également fallu `opens` certains packages (comme `com.gestionevents.model` à `com.fasterxml.jackson.databind` et `javafx.base`, et `com.gestionevents.g` à `javafx.fxml`) pour permettre la réflexion nécessaire à Jackson et à l'injection FXML.

— **Chargement des Fichiers FXML (`IllegalStateException: Location is not set.`)** : Cette erreur s'est produite car le `FXMLLoader` ne trouvait pas le fichier `MainView.fxml`. Il a fallu s'assurer que le chemin `/fxml/MainView.fxml` était correct et que le dossier `src/main/resources` était bien configuré comme dossier de ressources dans le build Maven, afin que les fichiers FXML soient

copiés dans `target/classes`.

— **`NullPointerException` dans le Contrôleur** : Une `NullPointerException` est survenue dans `MainViewController` car `filteredEventList` était utilisée avant d'être initialisée. L'ordre des opérations dans la méthode `initialize()` a dû être ajusté pour s'assurer que toutes les listes et liaisons nécessaires étaient en place avant leur utilisation.

— **Liaison des Données et Mise à Jour de l'UI** : Configurer correctement les `PropertyValueFactory` pour les colonnes de la `TableView`, gérer la mise à jour des détails de l'événement lors de la sélection, et assurer la réactivité de l'interface (par exemple, désactiver le bouton d'inscription) ont nécessité une attention particulière. L'utilisation de `Platform.runLater()` a été nécessaire pour les mises à jour de l'UI initiées depuis des threads non-JavaFX (bien que dans ce cas, la plupart des mises à jour soient sur le thread JavaFX).

3 Tests et Validation

Pour assurer la qualité et la fiabilité du système, une stratégie de tests unitaires a été adoptée.

— **Outils** : JUnit 5 a été utilisé comme framework de test, et Mockito pour la création d'objets mock (simulacres), notamment pour simuler le `NotificationService` dans les tests d'`Evenement`.

— **Cas de Test Couverts (Exemples)** :

  — `EvenementTest` :

    — Inscription et désinscription d'un participant.

    — Tentative d'inscription d'un participant déjà inscrit (`ParticipantDejaInscritExce`

    — Tentative d'inscription lorsque la capacité maximale est atteinte (`CapaciteMaxAtteint`

    — Vérification que l'annulation d'un événement notifie bien les participants inscrits (en utilisant un `NotificationService` mocké pour vérifier les appels à `envoyerNotification`).

    — Vérification que la modification d'un événement notifie les participants.

  — `GestionEvenementsTest` :

    — Ajout d'un nouvel événement.

    — Tentative d'ajout d'un événement avec un ID existant (`EvenementDejaExistantExce`

    — Suppression d'un événement existant et non existant (`EvenementNonTrouveExcepti`

    — Recherche d'événements.

— **Objectif de Couverture** : L'objectif initial était une couverture de test minimale de 70%, comme spécifié dans les livrables du TP. Les tests écrits ciblent les logiques

métier critiques.

— **Défis des Tests** :
  — **Test du Singleton (`GestionEvenements`)** : Tester un Singleton peut être délicat car son état persiste entre les tests. Pour assurer l'isolation des tests, une méthode de "nettoyage" manuel a été ajoutée dans la méthode `@BeforeEach` de `GestionEvenementsTest` pour vider la liste des événements avant chaque test, simulant ainsi une instance fraîche. Une approche alternative aurait été d'utiliser l'injection de dépendances au lieu d'un Singleton strict, ou d'implémenter une méthode `resetInstance()` dédiée (potentiellement via réflexion).

La sérialisation/désérialisation a été testée manuellement via l'interface graphique (sauvegarde et chargement des données) et par l'inspection du fichier JSON généré. Des tests unitaires plus poussés pour la sérialisation pourraient vérifier la conformité du JSON et la reconstruction correcte des objets polymorphiques.

4 Structure du Projet

Le projet est organisé en packages Maven suivant une structure logique pour séparer les préoccupations :

```
src/main/java/
|-- com/gestionevents/
|   |-- controller/
|   |   \-- GestionEvenements.java
|   |-- exceptions/
|   |   |-- CapaciteMaxAtteinteException.java
|   |   |-- EvenementDejaExistantException.java
|   |   |-- EvenementNonTrouveException.java
|   |   \-- ParticipantDejaInscritException.java
|   |-- gui/
|   |   |-- MainGuiApp.java
|   |   \-- controllers/
|   |       |-- EventEditDialogController.java
|   |       \-- MainViewController.java
|   |-- model/
|   |   |-- Concert.java
|   |   |-- Conference.java
```

```
|   |   |-- Evenement.java
|   |   |-- Intervenant.java
|   |   |-- Organisateur.java
|   |   \-- Participant.java
|   |-- observer/
|   |   |-- EvenementObservable.java
|   |   \-- ParticipantObserver.java
|   |-- services/
|   |   |-- EmailNotificationService.java
|   |   \-- NotificationService.java
|   \-- utils/
|       \-- SerialisationUtil.java

src/main/resources/
|-- fxml/
|   |-- EventEditDialog.fxml
|   \-- MainView.fxml
|-- styles.css
\-- module-info.java (Placé à la racine de src/main/java, mais conceptuel

src/test/java/
\-- com/gestionevents/
|-- controller/
|   \-- GestionEvenementsTest.java
\-- model/
\-- EvenementTest.java
```

Le fichier `pom.xml` à la racine du projet gère les dépendances (JavaFX, Jackson, JUnit, Mockito) et la configuration du build Maven, y compris le plugin JavaFX.

Conclusion

Ce projet a été une expérience d'apprentissage très complète, permettant de mettre en pratique de nombreux concepts fondamentaux et avancés de la Programmation Orientée Objet. La conception d'un système de gestion d'événements, depuis la modélisation du backend jusqu'à l'implémentation d'une interface graphique fonctionnelle, a couvert un large spectre de problématiques de développement logiciel.

Les fonctionnalités clés, telles que la gestion polymorphique des événements, le système de notification basé sur le pattern *Observer*, la gestion personnalisée des exceptions, et la persistance des données avec *Jackson*, ont toutes été implémentées avec succès. L'adhésion aux principes SOLID a guidé les choix de conception vers une architecture plus robuste et maintenable.

Les défis majeurs rencontrés, notamment la configuration du polymorphisme pour la sérialisation JSON et la mise en place de l'environnement JavaFX avec le système de modules Java, ont été surmontés et ont constitué des opportunités d'approfondir la compréhension de ces technologies. Le processus itératif de débogage et de résolution des erreurs (comme les `NullPointerException` ou les `IllegalStateException` lors du chargement FXML) a également été très formateur.

Apprentissages Clés :

— Importance d'une modélisation OO soignée en amont.
— Puissance des *design patterns* pour résoudre des problèmes récurrents.
— Nécessité d'une bonne gestion des exceptions pour la robustesse.
— Complexité et solutions pour la sérialisation d'objets polymorphiques.
— Spécificités de JavaFX et du système de modules Java.

Pistes d'Améliorations Futures :

— **Interface Utilisateur plus riche** : Amélioration du design CSS, ajout d'icônes, validation des entrées en temps réel.
— **Gestion des Utilisateurs plus avancée** : Authentification, rôles distincts (participant, organisateur, administrateur) avec des vues et des droits différents.
— **Base de Données Réelle** : Remplacer la sérialisation JSON par une intégration avec une base de données (SQL ou NoSQL) pour une persistance plus performante et évolutive.
— **Plus de Types d'Événements** : Étendre le système pour gérer d'autres types d'événements (ateliers, expositions, etc.).
— **Système de Paiement** : Intégrer une fonctionnalité de paiement pour les événements payants.
— **Tests plus Exhaustifs** : Augmenter la couverture des tests unitaires, ajouter des tests d'intégration pour l'interaction entre les composants, et potentiellement des tests UI.

En définitive, ce projet a permis de consolider les acquis en POO et de développer des compétences pratiques en conception et implémentation de logiciels.