Proposé par : **ELOUNDOU NGOUMA Thomas**

Matricule **: 22P294**

Niveau **: 3GI**

INTRODUCTION

Dans le cadre des Travaux Pratiques finaux de l'Unité d'Enseignement intitulée « Programmation Orientée Objet 2 » (POO 2), en troisième année de la filière Génie Informatique de l'ENSPY, il a été proposé un mini-projet portant sur la conception et l'implémentation d'un système de gestion d'événements distribué. Ce projet visait à mettre en application les connaissances acquises tout au long des enseignements de POO, notamment sur les notions d'héritage, de polymorphisme, de classes, d'interfaces, des collections, ainsi que sur les principes SOLID. Il avait également pour objectif l'introduction et la maîtrise de concepts plus avancés tels que les Design Patterns, la sérialisation/désérialisation de données (notamment en JSON), la programmation événementielle et la programmation asynchrone. Le système à développer devait permettre de gérer différents types d'événements (tels que des concerts ou des conférences), avec la possibilité pour les utilisateurs de s'inscrire ou se désinscrire, pour les organisateurs de gérer leurs événements, et pour les participants de recevoir des notifications en temps réel. Il était également impératif d'assurer la persistance des données, permettant ainsi leur sauvegarde et leur récupération automatique. Le présent rapport expose les décisions de conception adoptées durant le développement du projet. Il s'articule autour de plusieurs axes : une analyse du besoin fonctionnel, une présentation de l'architecture logicielle choisie, une modélisation orientée objet via des diagrammes UML, puis une synthèse des choix techniques opérés et leur justification.

I. ANALYSE DU BESOIN FONCTIONNEL

Le besoin fonctionnel du projet est centré sur la conception et la mise en œuvre d'un système distribué de gestion d'événements permettant à différents types d'utilisateurs (organisateurs et participants) de gérer efficacement des événements (conférences, concerts) à travers une interface intuitive.

Créer une application Java complète et graphique permettant :

- La **gestion centralisée** de plusieurs types d'événements
- La **gestion des inscriptions** de participants
- La **notification automatique** des changements liés aux événements
- La **sauvegarde persistante** des données

Les utilisateurs

- **Organisateur**
- Crée et configure des événements
- Définit la capacité maximale
- Peut annuler un événement
- Notifie les participants automatiquement
- **Participant**
    - Peut s'inscrire ou se désinscrire d'un événement
    - Reçoit des notifications (ex. annulation, changement de lieu/date)
    - Visualise les détails des événements
    -

Fonctions principales à implémenter

Fonction	Description
Créer un événement	L'organisateur choisit un type (conférence ou concert) et fournit ses paramètres
Afficher les événements	L'utilisateur voit tous les événements disponibles, avec leurs détails
Inscrire un participant	Ajoute un participant à la liste si la capacité n'est pas atteinte
Annuler un événement	Supprime un événement et notifie tous les inscrits
Notifier les participants	Système de notifications basé sur le pattern Observer
Sauvegardebles événements	Sérialisation automatique en JSON avec Jackson
Charger les événements	Récupération automatique des événements au lancement

Contraintes fonctionnelles

- Un événement a une **capacité maximale** fixe, au-delà de laquelle l'inscription est refusée.
- Un participant **ne peut pas s'inscrire deux fois** au même événement.
- Les notifications doivent être **immédiates et automatiques** dès qu'un événement est modifié.
- Les données doivent être **persistées entre les sessions**.

II. ARCHITECTURE LOGICIELLE CHOISIE

1. Architecture générale : modèle en 3 couches (3-Tiers)

Le système repose sur une architecture **modulaire en trois couches**, assurant une bonne séparation des responsabilités et facilitant la maintenance ainsi que l'extension du projet :

• Couche Présentation

- **Rôle** : Gérer les interactions avec l'utilisateur (console ou interface graphique JavaFX).
- **Composants** : Menus, formulaires, listes d'événements, vues JavaFX (FXML).
- **Communication** : Appelle les services de la couche métier (ex. : `GestionEvenements.ajouterEvenement()`).

• Couche Métier (Logique de traitement)

- **Rôle** : Implémente les règles métier et coordonne les opérations applicatives.

- **Composants principaux** :
    - **Classes métier** : `Evenement`, `Conference`, `Concert`, `Participant`, `Organisateur`
    - **Services** :
        - `GestionEvenements` (Singleton) : centralise la gestion de tous les événements.
        - `NotificationService` : gère l'envoi des notifications (basé sur le pattern **Observer**).
    - **Exceptions personnalisées** :
        - `CapaciteMaxAtteinteException`
        - `EvenementDejaExistantException`

• Couche Persistance

- **Rôle** : Sauvegarde et chargement des données à travers la sérialisation.
- **Technologies** :
    - **Sérialisation JSON** via la bibliothèque **Jackson**
    - Utilisation de **collections Java** telles que `Map<String, Evenement>` pour accéder efficacement aux événements par identifiant

2. Exemple de flux de données : Annulation d'un événement

Le processus d'annulation d'un événement illustre bien l'interaction entre les couches :

1. **Couche Présentation**
   L'organisateur déclenche l'annulation via l'interface (bouton, commande...).
2. **Couche Métier**
   La méthode `GestionEvenements.annulerEvenement()` est appelée :
    - Elle met à jour l'état de l'événement (ex. : le retirer de la liste).
    - Elle déclenche une notification en appelant le `NotificationService`.
3. **NotificationService**
    - Il parcourt la liste des participants inscrits à l'événement.
    - Il envoie une notification personnalisée à chacun (console ou fenêtre graphique).

III. CONCEPTION ORIENTÉE OBJET

Diagramme de Classes

Objectif : Modéliser la structure statique du système, incluant les classes, attributs, méthodes, et relations.

Éléments clés :

- Classes principales :
  - `Evenement` (abstraite) : Contient les propriétés de base (`id`, `nom`, `date`, etc.) et méthodes communes (`annuler()`, `afficherDetails()`).
  - `Conference`/`Concert` : Héritent de `Evenement` et ajoutent des attributs spécifiques (`theme`, `artiste`).
  - `Participant` et `Organisateur` : Gestion des utilisateurs avec héritage.
- Relations :
  - **Héritage** (flèche pleine) :
    - `Evenement` → `Conference` et `Concert`.
    - `Participant` → `Organisateur`.
  - **Association** (ligne simple) :
    - `Participant` est associé à `Evenement` (inscription).
  - **Dépendance** (ligne pointillée) :
    - `NotificationService` dépend de `Participant` pour envoyer des notifications.Utilise `envoyerNotification()` pour chaque participant (synchrone ou asynchrone avec `CompletableFuture`).

1. **Couche Persistance** :
   - Sauvegarde les modifications dans un fichier JSON/XML.

3. Principes Architecturaux Clés

- **Découpage modulaire** :
  - Chaque couche a une responsabilité unique (ex: la couche métier ignore comment les données sont stockées).
- **Faible couplage** :

    - Les composants communiquent via des interfaces (ex: `NotificationService`).
- **Extensibilité** :
    - Ajout facile de nouveaux types d'événements (ex: `Atelier` héritant de `Evenement`).
- **Gestion des erreurs** :
    - Exceptions métier pour clarifier les problèmes (ex: capacité maximale atteinte).
        -

IV.CHOIX TECHNIQUES ET JUSTIFICATION

Composant	Choix Technique	Justification
Gestion des événements	Singleton (`GestionEvenements`)	Garantit un point d'accès unique à la liste des événements, évitant les incohérences.
Notifications	Pattern **Observer**	Découple l'émetteur (événement) des récepteurs (participants).
Sérialisation	JSON avec `Jackson`	Léger, lisible, et largement supporté. Facilite le débogage.
Programmation async	`CompletableFuture`	Améliore les performances pour les notifications sans bloquer le thread principal.

---

5. Scénarios d'Utilisation

- **Création d'un événement** :
    - L'organisateur remplit un formulaire → `GestionEvenements.ajouterEvenement()` → Sauvegarde en JSON.
- **Inscription d'un participant** :
    - Vérification de la capacité → Notification de confirmation.
- **Annulation** :
    - Notification en temps réel à tous les participants.

CONCLUSION

En somme, la réalisation de ce projet a constitué une étape importante dans notre apprentissage de la programmation orientée objet et des bonnes pratiques de conception logicielle. En développant un système complet de gestion d'événements, nous avons non seulement appliqué des notions fondamentales comme l'héritage, le polymorphisme et l'utilisation des collections, mais aussi intégré des concepts plus avancés tels que les design patterns, la sérialisation de données et la programmation événementielle. Ce projet nous a permis de mieux comprendre l'importance d'une architecture logicielle bien pensée, de la modularité du code et de l'organisation d'un projet selon les standards professionnels (comme Maven et JavaFX). Au-delà de l'aspect technique, il a également renforcé notre capacité à structurer une solution complète, de l'analyse du besoin jusqu'à l'interface utilisateur fonctionnelle. Il représente ainsi une synthèse concrète de nos acquis en POO et une base solide pour aborder des problématiques logicielles plus complexes à l'avenir.