RÉPUBLIQUE DU CAMEROUN
UNIVERSITÉ DE YAOUNDÉ I
ÉCOLE NATIONALE SUPÉRIEURE POLYTECHNIQUE DE YAOUNDÉ

PROGRAMMATION ORIENTÉE OBJET II

RAPPORT EXPLICATIF TP FINAL

MBIAKE Emmanuella Rose 24P756

Supervisé par :
Dr. **KUNGNE**

3^{e} Année - Génie Informatique

Anné scolaire : 2024-2025

1 Introduction

Avant l'implémentation et la mise sur pied de tout système d'information, il est important que celui-ci soit modélisé et étudié pour que l'on puisse prendre connaissance des différentes classes intervenantes, des méthodes, et autres caractéristiques essentielles pour être sûr de produire un résultat qui saura répondre aux attentes présentées. Pour ce faire, nous allons commencer par faire une description textuelle du projet, sur laquelle l'on pourra s'appuyer pour faire une modélisation UML efficace.

2 Modélisation des Classes (UML)

2.1 Description textuelle

On va considérer qu'on a un propriétaire qui a plusieurs salles et les loue à des personnes qui organisent des évènements. L'application est installée dans la machine du propriétaire et il est celui qui gère et manipule l'application.
Les évènements pris en compte sont les conférences et les concerts.
Pour avoir accès à une salle, un organisateur passe par le propriétaire. Et si les deux s'accordent, celui-ci fournit au propriétaire les horaires et la liste des participants.
Avant d'inscrire définitivement un partcipant, le propriétaire lui notifie par mail via l'application qu'on veut l'inscrire à un évènement. Et ce n'est que si il accepte qu'il est effectiement inscrit.
Un participant peut participer à un ou plusieurs évènements. Un participant reçoit des notifications le jour de l'évènement auquel il participe très tôt le matin.
Un évènement accueille un ou plusieurs participants. Un organisateur organise un ou plusieurs évènements.
Un organisateur peut ajouter, supprimer ou rechercher un ou plusieurs évènements sur la plateforme. Un participant peut rechercher un évènement sur la plateforme. Un organisateur peut définir un nombre de places maximal pour un évènement.
On peut ajouter un participant à un évènement, supprimer un participant d'un évènement, annuler un évènement, ou afficher les détails de l'évènement.

2.2 Différentes classes

Nous présenterons les classes avec les attributs et les méthodes correspondantes. Nous avons en tout huit classes métier.

2.2.1 Évènement

Cette classe est une classe abstraite. Ses attributs sont :

- id: String;
- nom: String;
- date: LocalDateTime;
- lieu: String;
- capaciteMax: int;
- participants: List¡Participant¿.

Ses méthodes sont :

- ajouterParticipant(p: Participant): void;
- annuler(): void;
- afficherDetails(): void.

2.2.2 Concert

Cette classe hérite de la classe "Evenement". Ses attributs propres sont :

- artiste: String;
- genreMusical: String.

Ses méthodes propres sont :

- afficherDetails(): void.

2.2.3 Conférence

Cette classe hérite de la classe "Evenement". Ses attributs propres sont :

- theme: String;
- intervenants: List¡Intervenant¿.

Ses méthodes propres sont :

- afficherDetails(): void.

2.2.4 Participant

Cette classe est une classe abstraite. Ses attributs sont :

- id: String;
- nom: String;
- email: String.

Ses méthodes sont :

- recevoirNotification(msg: String): void.

2.2.5 Organisateur

Cette classe hérite de la classe "Participant". Ses attributs propres sont :

- evenementsOrganises: List¡Evenement¿.

Ses méthodes propres sont :

- ajouterEvenement(e: Evenement): void

2.2.6 Intervenant

Cette classe hérite de la classe "Participant". Ses attributs propres sont :

- domaineIntervention: String.

Ses méthodes propres sont :

- recevoirNotification(msg: String): void

2.2.7 GestionEvenement

Ses attributs sont :

- instance: GestionEvenements (static);
- evenements: Map¡String, Evenement¿.

Ses méthodes sont :

- getInstance(): GestionEvenements;
- ajouterEvenement(e: Evenement): void;
- supprimerEvenement(id: String): void;
- rechercherEvenement(id: String): Evenement.

2.3 Éléments d'Implémentation

2.3.1 NotificationService

Il s'agit d'une interface. L'attribut statique qu'on y trouve est "message : String". La méthode qu'on y trouve est : "envoyerNotification(msg: String): void".

2.4 Utilisation du Design Pattern Observer

Le design pattern Observer a pour but de permettre à un objet sujet tel que "Evenement" par exemple, de notifier automatiquement tous ses observateurs à l'exemple de "Participant" quand il y a un changement.
J'ai ajouté ce programme dans mon code en quatre étapes.

- Créer les interfaces "ParticipantObserver" que tous les participants implémenteront et "EvenementObservable" que tous les évènements implémenteront;
- Modifier "Participant" pour implémenter "ParticipantObserver" et "Evenement" pour implémenter "EvenementObservable";
- Ajouter une liste d'observateurs dans Evenement;
- Créer la classe "TestObserver" pour tester.

2.5 Gestion des Exceptions Personnalisées

Je l'ai fait en trois étapes principales :

- Créer les classes "CapaciteMaxAtteinteException" et "EvenementDejaExistantException";
- Les utiliser dans mon coe, notamment dans "Evenement.ajouterParticipant()" et "GestionEvenements.ajouterEvenemen
- Créer une classe "TestExceptions" pour tester.

2.6 Sérialisation JSON/XML

J'ai fait ceci en quatre étapes principalement :

- Ajouter dans la section ¡dependencies¿, les dépendances "jackson-databind" et "jackson-datatype-jsr310";
- Créer la classe "JsonUtils";
- Ajouter les annotations nécessaires dans la classe "Evenement";
- Créer une "classe TestJson" pour tester.

2.7 Programmation Asynchrone

3 Tests et Validation

J'ai écrit des tests "JUnit" pour cinq de mes classes.