Rapport de Projet – TP3 POO

Conception d'un Système de Gestion d'Événements Distribué

Nom : MBO'O ATENA SIDONIE ORNELLA
Matricule : 24P749
Encadrant : W. Kungne

26 mai 2025

Table des matières

1 Introduction

Dans le cadre de ce troisième TP de Programmation Orientée Objet (POO), il nous a été demandé de concevoir un système de gestion d'événements distribué. Ce projet individuel a pour objectif de mettre en pratique les concepts avancés de la POO en Java, notamment l'héritage, le polymorphisme, les interfaces, les exceptions personnalisées, ainsi que plusieurs design patterns tels que Singleton, Observer, Strategy et Factory.

L'application permet de gérer différents types d'événements (conférences, concerts, etc.), d'inscrire des participants, de gérer des organisateurs, d'envoyer des notifications et de persister les données avec sérialisation. Ce rapport présente les outils utilisés, la modélisation UML, le modèle objet, les tests, et les choix de conception.

2 Outils utilisés pour réaliser le projet

2.1 Environnement de développement

Ce projet a été développé principalement sous l'environnement **IntelliJ IDEA** avec le SDK Java 11.

FIGURE 1 – IDE IntelliJ IDEA

2.2 Outils de gestion de version

Le projet a été versionné avec **Git**, hébergé sur **GitHub**, facilitant le suivi des modifications et la sauvegarde du code.

2.3 Bibliothèques utilisées

Pour la sérialisation JSON, la bibliothèque **Jackson** a été utilisée, ainsi que les API Java Collections pour la gestion des données.

2.4 Outils UML

Le diagramme UML a été conçu à l'aide d'outils comme **draw.io**.

FIGURE 2 – Dépôt GitHub

FIGURE 3 – Jackson - Sérialisation JSON

FIGURE 4 – Exemple de diagramme dans StarUML

2.5 Outils de test

Les tests unitaires ont été réalisés à l'aide de **JUnit 5** pour assurer la stabilité et la robustesse du code.

FIGURE 5 – JUnit pour les tests unitaires

3 Diagrammes UML

3.1 Diagramme des classes

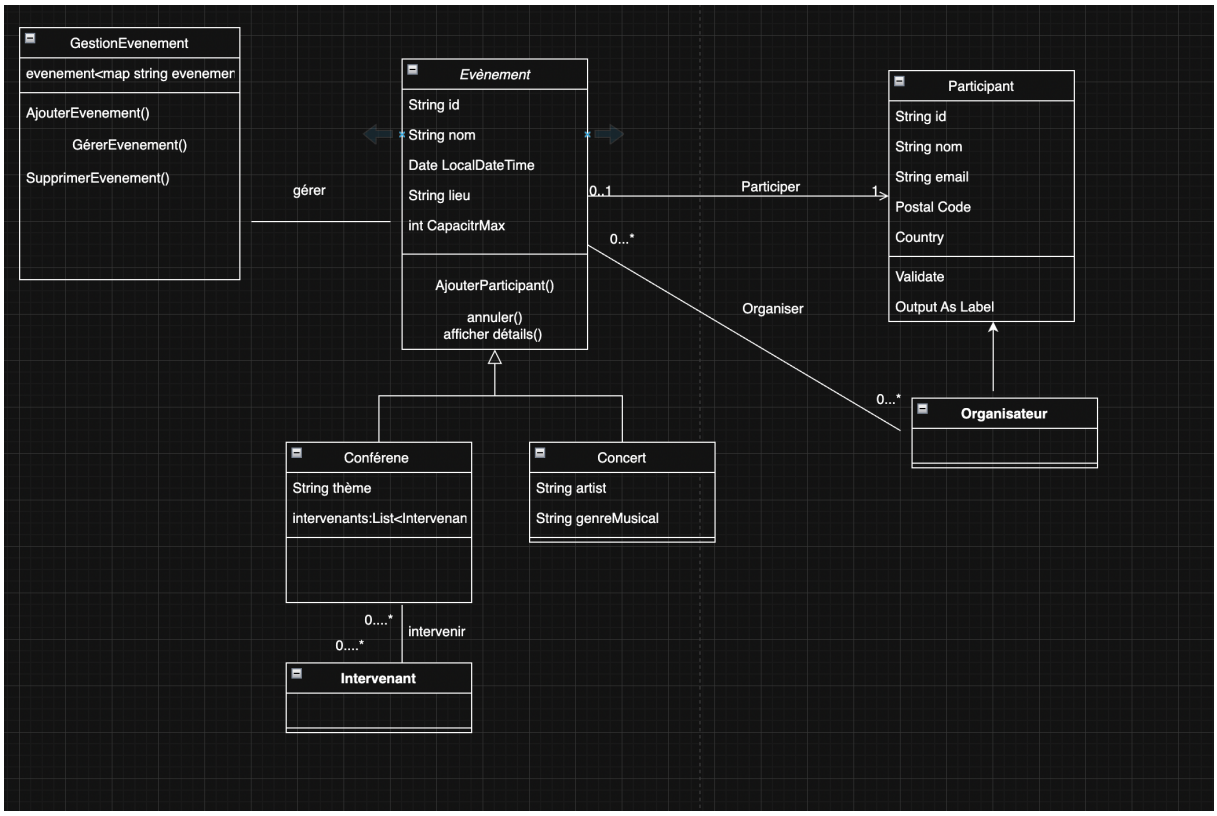

FIGURE 6 – Diagramme de classes

3.2 Diagramme des cas d'utilisation

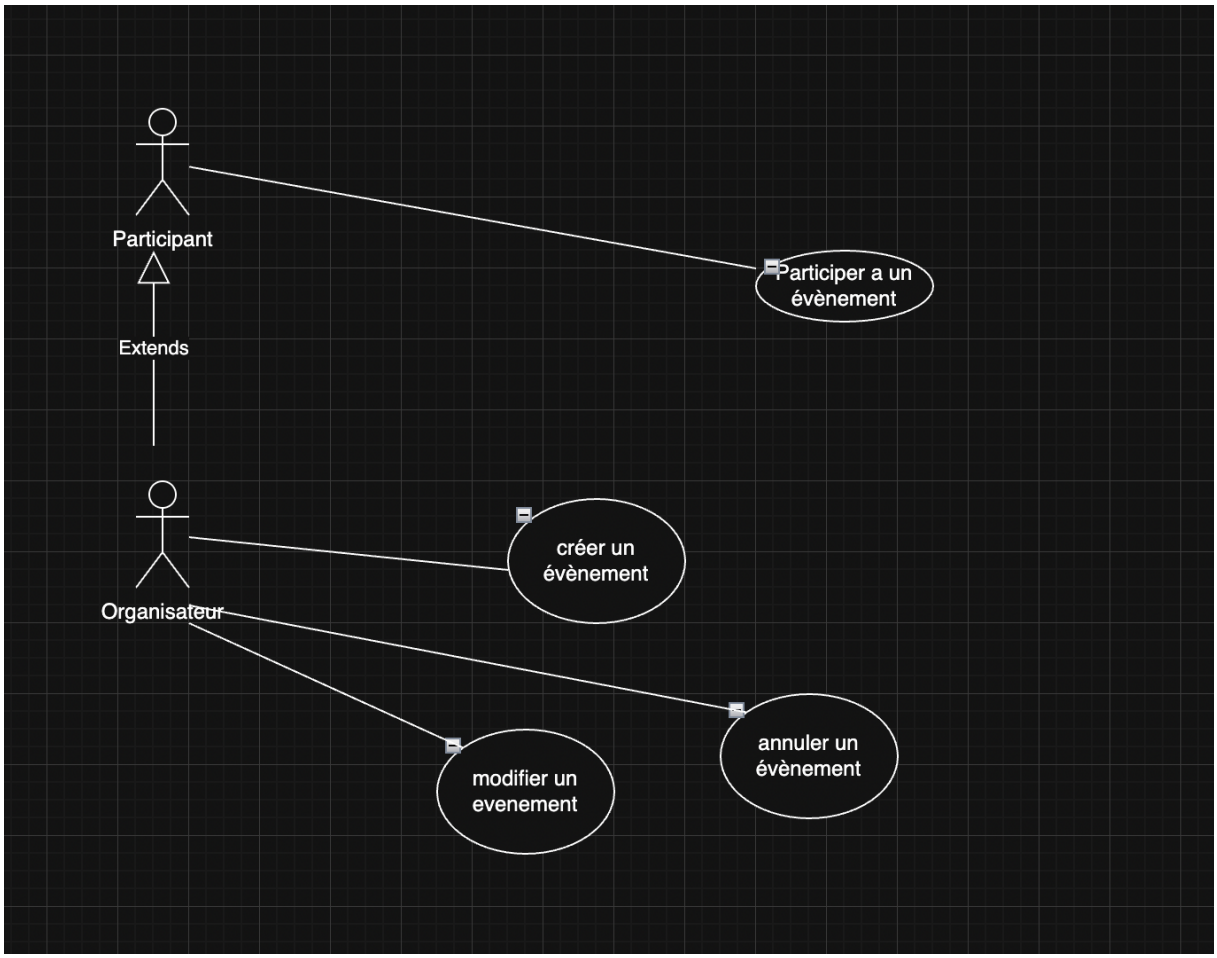

FIGURE 7 – Diagramme des cas d'utilisation

3.3 Diagramme de package

FIGURE 8 – Diagramme de séquence

4 Eléments d'implémentation

4.1 Modèle de données

Le modèle utilisé ici est le modèle mvc`Evenement`, étendue par `Conference` et `Concert`.

4.2 modèle

Représente les données, les règles métiers, et la logique métier de l'application.
Gère les données de l'application (par exemple : événements, participants...).
Ne contient aucune logique de présentation.
Peut notifier les vues quand il est mis à jour (souvent via le pattern Observer).

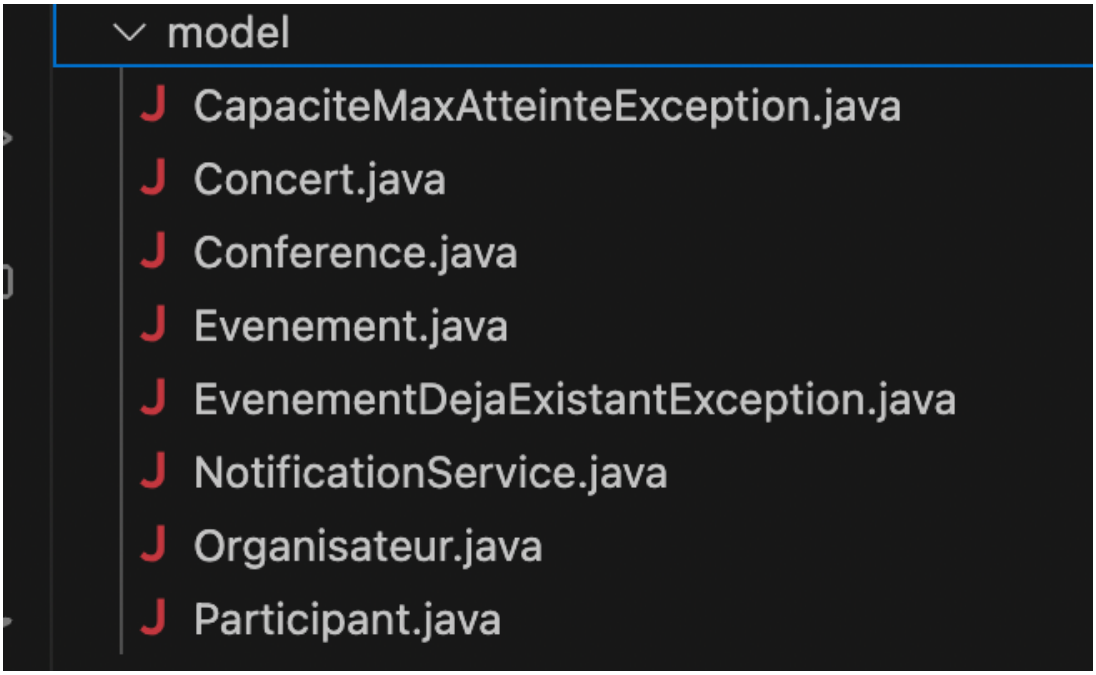

FIGURE 9 – Modèle de notification (Observer)

4.3 La vue

Représente l'interface utilisateur : ce que voit et manipule l'utilisateur.
Affiche les données fournies par le modèle.
Ne contient aucune logique métier.
Peut proposer des composants graphiques, formulaires, boutons...

4.4 Le controller

Joue le rôle d'intermédiaire entre le modèle et la vue.
Reçoit les entrées utilisateur depuis la vue.
Interagit avec le modèle pour modifier les données.
Peut aussi mettre à jour la vue si besoin.

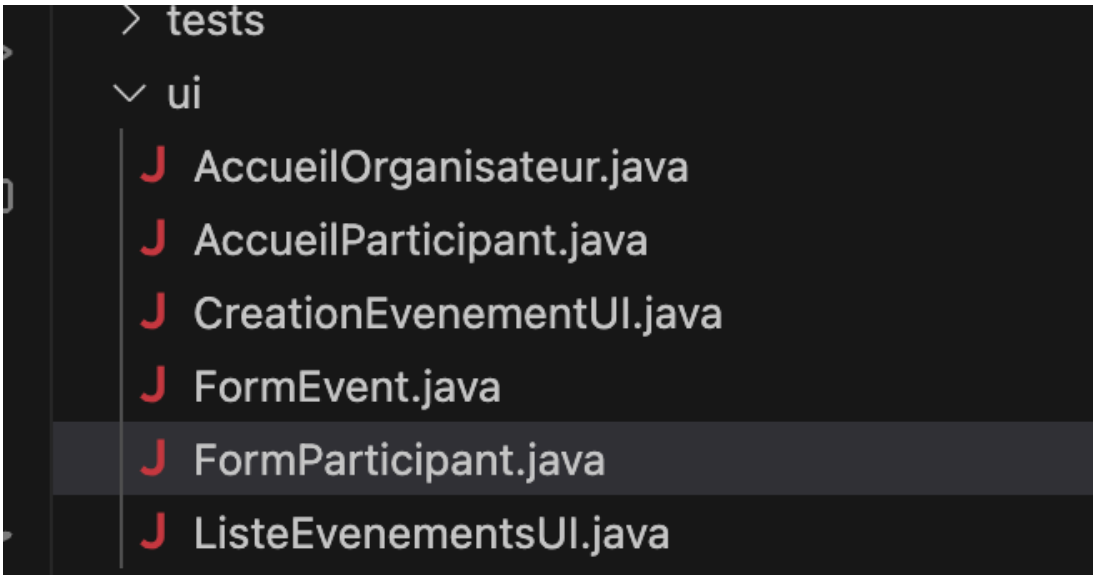

FIGURE 10 – La vue

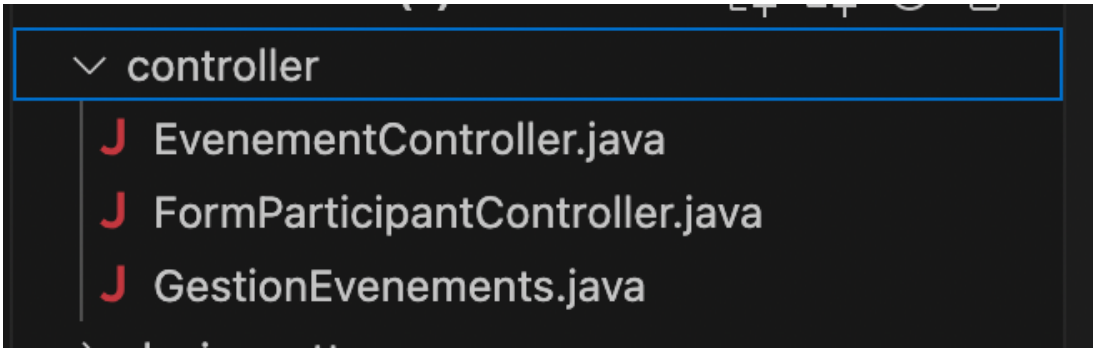

FIGURE 11 – Le controller

4.5 Gestion des Exceptions Personnalisées

Dans ce projet, nous avons défini des exceptions spécifiques pour mieux gérer les cas particuliers rencontrés lors de l'inscription ou la gestion des événements. Cela permet une gestion fine des erreurs, améliore la lisibilité du code et renforce la robustesse de l'application.

— **CapaciteMaxAtteinteException** : levée lorsque le nombre maximal de participants à un événement est atteint.

— **EvenementDejaExistantException** : utilisée pour éviter l'ajout de doublons d'événements.

4.6 Sérialisation JSON/XML

La sérialisation permet de sauvegarder les objets Java dans un fichier JSON. Pour cela, la bibliothèque **Jackson** a été utilisée.

— `ObjectMapper.writeValue(...)` : pour la sauvegarde dans un fichier JSON.

— `ObjectMapper.readValue(...)` : pour recharger les objets depuis un fichier JSON.

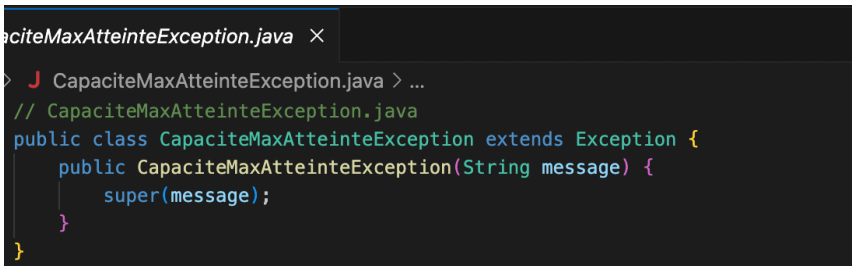

FIGURE 12 – Exception pour capacité maximale atteinte

FIGURE 13 – Exception pour événement déjà existant

FIGURE 14 – Exemple de sérialisation JSON

FIGURE 15 – Exemple de désérialisation JSON

4.7 Utilisation de Streams et Lambdas

L'utilisation des Streams et expressions lambda a permis de rendre le code plus clair et fonctionnel.

— **Filtrage des événements à venir**

```
evenements.stream()
  .filter(e -> e.getDate().isAfter(LocalDateTime.now()))
  .collect(Collectors.toList());
```

— **Extraction des emails des participants**

```
participants.stream()
  .map(Participant::getEmail)
  .forEach(System.out::println);
```

5 Tests et validation

Les tests ont été réalisés avec JUnit 5. Ils couvrent :

— L'ajout et la suppression de participants à un événement.

— La gestion des exceptions comme `CapaciteMaxAtteinteException`.

— La sérialisation et désérialisation des événements.

— Les notifications envoyées lors de l'annulation d'un événement.

Une couverture de test supérieure à 70% a été obtenue grâce à l'outil intégré de couverture dans IntelliJ IDEA.

6 Conclusion

Ce projet a permis d'approfondir la maîtrise de la programmation orientée objet avec Java. L'usage des design patterns (Observer, Singleton, Factory), la manipulation des collections, et la gestion d'exceptions personnalisées ont été des aspects enrichissants. Le système conçu est extensible, testable et répond aux exigences fonctionnelles posées en début de TP. Le respect de bonnes pratiques de modélisation UML et de structuration du code a également été une priorité.

7 Références

Références

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[2] Oracle. *The Java™ Tutorials*. Disponible sur : `https://docs.oracle.com/javase/tutorial/`

[3] Geary, David. *Core JSTL : Mastering the JSP Standard Tag Library*. Chapitre sur le modèle MVC.

[4] FasterXML. *Jackson JSON Processor*. Disponible sur : `https://github.com/FasterXML/jackson`

[5] Wikipedia contributors. *Observer pattern*. Disponible sur : `https://en.wikipedia.org/wiki/Observer_pattern`