Ecole Nationale Supérieure Polytechnique de Yaoundé

Département de génie Informatique

POO 2

PROJET FINAL DE POO : **GESTION DES EVENEMENTS**

NOMS :**KENMEUGNE TCHOUNGA MICHELE ESTELLE**
MATRICULE :**22P216**
CLASSE :**3 GI**
ANNEE :**2024-2025**

Enseignants Superviseur :
Pr BATCHAKUI
Dr KUNGNE

27 mai 2025

Table des matières

Introduction

Ce document présente le système de gestion d'événements développé dans le cadre du module de Programmation Orientée Objet avancée. L'application permet d'administrer différents types d'événements tels que des conférences et des concerts, avec des fonctionnalités complètes d'inscription et de notification des participants.

1 Objectifs du Projet

Le système devait répondre aux exigences fonctionnelles suivantes :

- Gestion centralisée des événements (création, modification, suppression)
- Inscription et suivi des participants
- Notification automatique des changements
- Persistance des données entre les sessions

Sur le plan technique, les principaux objectifs étaient :

- Mettre en œuvre les concepts d'héritage et de polymorphisme
- Utiliser des design patterns appropriés (Observer, Singleton)
- Implémenter une gestion robuste des erreurs
- Assurer la sérialisation/désérialisation des données

2 Conception Architecturale

2.1 Structure Globale

L'architecture suit le modèle en couches suivant :

- **Couche Métier** : Contient les classes fondamentales (Evenement, Conference, Concert, Participant)
- **Couche Accès aux Données** : Gère la persistance (sérialisation JSON)
- **Couche Présentation** : Interface en ligne de commande pour les tests

2.2 Design Patterns Implémentés

2.2.1 Singleton

Le pattern Singleton a été choisi pour la classe `GestionEvenements` afin de garantir qu'une seule instance gère l'ensemble des événements. Cette approche présente plusieurs avantages :

- Accès cohérent à la liste des événements
- Évite les duplications de données
- Centralise les opérations de gestion

```
1
2  public class GestionEvenements {
3      private static GestionEvenements instance;// instance unique du
       singleton
4      protected Map<String, Evenement> evenements;// collection d'
       evenements
5      private SerializeJson serializer;// c'est un nouvel attribut
6
7      // POur le constructeur priv
8      private GestionEvenements() {
9          evenements = new HashMap<>();//
10 // cela signifie que lorsqu'on cree l'unique instance de
       GestionEvenement, on pr pare un espace m moire pour stocker les
        venemnts
11         serializer=new SerializeJson();// initialisation
12     }
13
14     // L'acc s    l'instance unique
15     public static GestionEvenements getInstance() {
16         if (instance == null) {
17             instance = new GestionEvenements();
18         }
19         return instance;
20     }
21
22 }
```

2.2.2 Observer

Le pattern Observer permet une communication efficace entre les événements et les participants. Sa mise en œuvre comprend :

- L'interface `EvenementObservable` pour les sujets observables
- L'interface `ParticipantObserver` pour les observateurs
- Un mécanisme de notification automatique

```
1  public interface EvenementObservable {
2
3      // M thode pour s'abonner
4      public void ajouterParticipant(ParticipantObserver observer) throws
       CapaciteMaxAtteinteException;
5      // POur retirer
6      public void retirerObserver(ParticipantObserver observer);
7
8      // methode pour notifier tout le mondes
9      public void notifierObservers(String message);
10 }
11
12 }
```

Pour ParticipantObservable, on aura

```
1  public class ParticipantObserver implements NotificationService{
2  private String email;
3  private List<EvenementObservable> evenementsDuParticipant=new ArrayList<
       EvenementObservable>();
4  private NotificationService notificationService = this;
```

```
    public ParticipantObserver(String email) {
        this.email = email;
    }

    public List<EvenementObservable> getEvenementsDuParticipant() {
        return evenementsDuParticipant;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public NotificationService getNotificationService() {
        return notificationService;
    }

    public void setNotificationService(NotificationService
   notificationService) {
        this.notificationService = notificationService;
    }

    @Override
    public void envoyerNotification(String message) {
        System.out.println("Notification pour "+ email + ":" +message);

    }
}
```

3 Persistance des Données

3.1 Choix Technologique

Le format JSON a été préféré à XML pour plusieurs raisons :

— Syntaxe plus concise et lisible

— Meilleure performance en sérialisation

3.2 Implémentation

La sérialisation est gérée par la bibliothèque Jackson, avec une configuration spécifique pour supporter les types complexes dont on a ajouté la dépendance :

```
public class SerializeJson {
    private final ObjectMapper mapper;
    // final car sa valeur ne change pas apr s l'initialisation
    //ObjetcMapper est la classe pricipale de Jackson pour la
   s rialisation

    public SerializeJson() {
        this.mapper = new ObjectMapper();
        mapper.registerModule(new JavaTimeModule());// Active le support
    des dates
```

```
9         mapper.enable(SerializationFeature.INDENT_OUTPUT);// rend Jason
  lisible
10
11     }
12
13     // M thode pour sauvegarder les Objet dans un fichier Json
14     public <T> void sauvegarderObjet(T objet, String cheminFichier)
  throws Exception {
15         mapper.writeValue(new File(cheminFichier), objet);
16
17     }
18     // writeValue(): convertit l'objet en JSON et l' crit  dans le
  fichier
19     // On utilise T pour tout type d'objet
20
21     // M thode pour sauvegarder une liste
22     public <T> void sauvegarderListe(List<T> objets, String
  cheminFichier) throws Exception {
23         mapper.writeValue(new File(cheminFichier), objets);
24
25     }
26
27     // M thode pour charger un Objet depuis un ficheir JSON
28
29     public <T> T chargerObjet(String cheminFichier, Class<T> classe)
  throws Exception {
30         return mapper.readValue(new File(cheminFichier),classe);
31
32     }
33     // classe sp cifie le type d'Objet    cr er
34
35
36     // M thode pour charger une liste
37
38     public <T> List <T> chargerListe(String cheminFichier,Class <T>
  classe) throws Exception{
39         return mapper.readValue(
40                 new File(cheminFichier),
41                 mapper.getTypeFactory().constructCollectionType(List.
  class, classe)
42         );
43     }
44     // constructCollectionTpype() sp cifie que le JSON contient une
  lsite d'objet de type class contient
45
46
47     // Nouvelle m thode pour ajouter    un fichier existant
48     public <T> void ajouterAListe(T element, String cheminFichier, Class
  <T> classe) throws Exception {
49         List<T> existants = new ArrayList<>();
50
51         // Lire le fichier existant s'il y en a un
52         if (Files.exists(Paths.get(cheminFichier))) {
53             existants = chargerListe(cheminFichier, classe);
54         }
55
56         // Ajouter le nouvel   lment
57         existants.add(element);
```

```
58
59          // R   crire le fichier complet
60          sauvegarderListe(existants, cheminFichier);
61      }
62  }
```

4 Validation du Système

4.1 Stratégie de Test

La validation du système s'est appuyée sur :

- Des tests unitaires pour chaque composant
- Des tests de charge et de sauvegarde pour la gestion des participants ;
- Des tests de robustesse pour la gestion des erreurs pour les classes **"CapaciteMaxAtteinteException** et **EvenementDejaExistantException**

4.2 Exemple de Test Unitaires

```
1  public class GestionEvenementTest {
2
3      // Pour faire la sauvegarde je fais dabord la s rilisation, De
       m me pour charger
4
5      @Test
6      void testSauvegarderConcert() throws Exception {
7          // 1) Cr ons un evenement
8          Concert concert1 = new Concert(
9                  "c1",
10                 "Concert de Gospel",
11                 LocalDateTime.of(2025, 5, 25, 20, 0),
12                 "Paris",
13                 20,
14                 "Morijah",
15                 "Gospel"
16
17         );
18         //2) ajoutons l'evenemt
19         GestionEvenements gestion = GestionEvenements.getInstance();
20         gestion.ajouterEvenement(concert1);
21         // 3- Sauvegarder dans le fichier JSON
22         String cheminFichier = "concert.json";
23         gestion.sauvegarderEvenement(cheminFichier);
24
25         //4) V rifier que le fichier est cr e et n'est pas vide
26         File fichier = new File(cheminFichier);
27         assertTrue(fichier.exists());
28         assertTrue(fichier.length() > 0);
29
30     }
31
32     @Test
33     void testSauvegarderConference() throws Exception {
34         // 1) Cr ons un evenement
```

```
        Conference conference1 = new Conference(
                "conf1",
                "Conference de Recyclahe",
                "Yaound ",
                5,
                LocalDateTime.of(2025, 4, 5, 9, 30)

        );
        //2) ajoutons l'evenemt
        GestionEvenements gestion = GestionEvenements.getInstance();
        gestion.ajouterEvenement(conference1);
        // 3- Sauvegarder dans le fichier JSON
        String cheminFichier = "conference.json";
        gestion.sauvegarderEvenement(cheminFichier);

        //4) V rifier que le fichier est cr e et n'est pas vide
        File fichier = new File(cheminFichier);
        assertTrue(fichier.exists());
        assertTrue(fichier.length() > 0);

    }


}
```

4.3 Exemple de Données JSON

Voici un exemple concret de fichier JSON généré par le système :

```
[
  {
    "type": "concert",
    "id": "c1",
    "nom": "Concert de Gospel",
    "lieu": "Paris",
    "artiste": "Morijah",
    "genreMusical": "Gospel",
    "date": "2025-05-25 20:00:00",
    "capacity": 20,
    "participants": []
  }
]
```

- `type` : Type d'événement ("concert" ou "conference") Il a été ajouté pour la désérialisation Etant donné qu'on aura besoin de recupérer un type d'Evenement Précis(entre Concert et Conférence)
- `id` : Identifiant unique (clé primaire)
- `capacity` : Capacité maximale en participants
- `participants` : Liste des inscrits (vide ici)

Conclusion

Ce projet a permis de développer un système complet de gestion d'événements mettant en œuvre les principes avancés de la programmation orientée objet. Les choix architectu-

raux, notamment l'utilisation des design patterns Singleton et Observer, se sont révélés pertinents pour répondre aux exigences fonctionnelles.
Les principaux défis techniques ont concerné :

— La gestion de la désérialisation des objets polymorphes

— La mise en œuvre efficace du pattern Observer

— La validation complète des cas d'erreur

Les perspectives d'évolution :

— L'ajout d'une interface graphique

— L'extension à de nouveaux types d'événements

— L'amélioration des performances pour de gros volumes

— Bonne Gestion de la Désérialisation

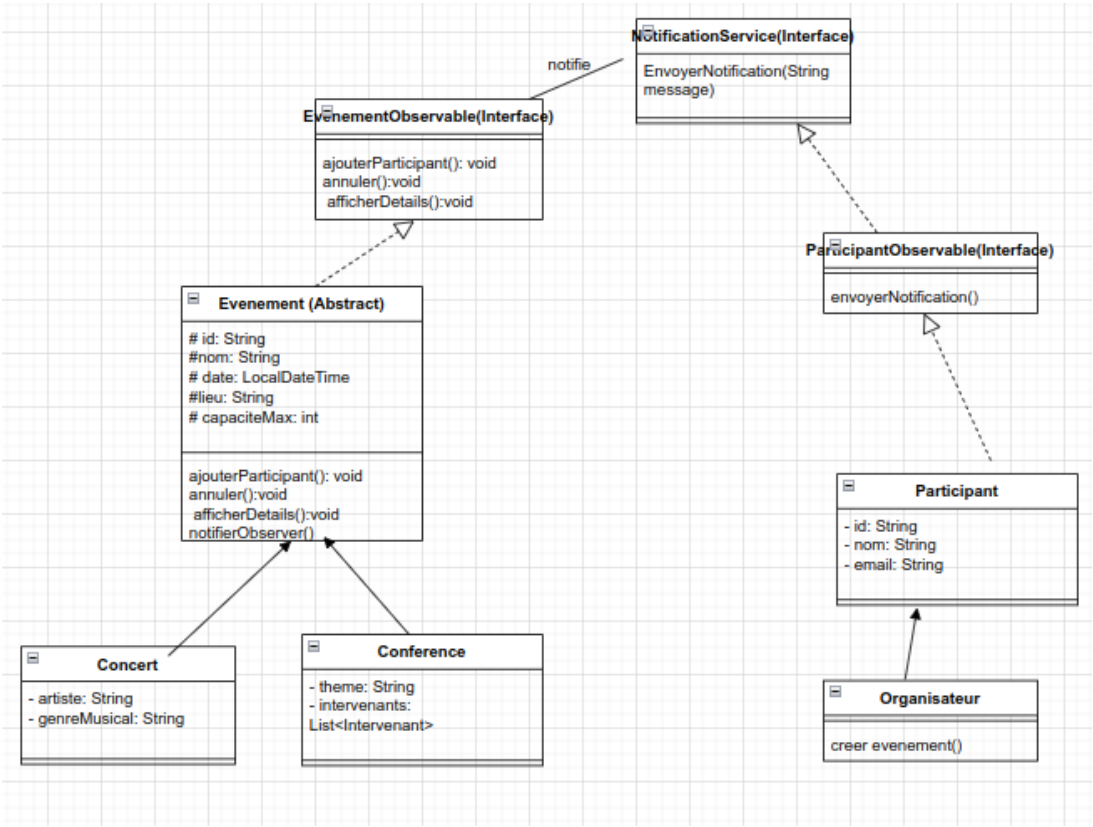

FIGURE 1 – Diagramme de classes simplifié du système