Système de Gestion d'Événements
Rapport TP#3

Projet de Programmation Orientée Objet

Réalisé par : HEUMI BIATEU ARTHUR FRESNEL

Date : 26 Mai 2025

ENSPY

Année académique 2024–2025

Table des matières

1 Introduction

L'objectif de ce projet, dans le cadre du TP#3 de Programmation Orientée Objet, était de concevoir et développer un système distribué de gestion d'événements. Ce système devait permettre la gestion complète d'événements (conférences, concerts, etc.), incluant la création, la modification, l'annulation, l'inscription des participants, et la sérialisation des données en formats JSON et XML. De plus, le projet demandait l'implémentation de plusieurs patrons de conception (Factory, Singleton, Observer, Strategy), une couverture de test d'au moins 70%, et l'utilisation de fonctionnalités avancées comme la programmation asynchrone et les streams/lambdas de Java.

Ce rapport détaille les différentes étapes du développement, les choix techniques effectués, les défis rencontrés, et les résultats obtenus. Il est structuré comme suit : une vue d'ensemble du projet, une description des patrons de conception utilisés, les détails d'implémentation, les tests réalisés, les défis rencontrés, et une conclusion.

2 Vue d'ensemble du projet

Le système de gestion d'événements est une application Java qui permet de gérer des événements de différents types (conférences et concerts), des participants, et des organisateurs. Voici les principales fonctionnalités implémentées :

— Création et gestion des événements via une classe singleton `GestionEvenements`.

— Support de différents types d'événements (conférences avec intervenants, concerts avec artistes).

— Inscription des participants avec notifications automatiques (patron Observer).

— Recherche d'événements par nom ou lieu avec les streams et lambdas.

— Sérialisation des données en JSON et XML.

— Tri des événements selon différents critères (patron Strategy).

— Gestion asynchrone des notifications.

2.1 Architecture du projet

Le projet est organisé en plusieurs packages pour une meilleure modularité :

— `com.evenements.model` : Contient les classes de modèle (`Evenement`, `Conference`, `Concert`, `Participant`, `Organisateur`, `Intervenant`).

— `com.evenements.service` : Inclut les services comme `GestionEvenements` et les services de notification.

— `com.evenements.factory` : Implémentation du patron Factory pour la création des objets.

— `com.evenements.observer` : Interfaces pour le patron Observer.

— `com.evenements.strategy` : Implémentation du patron Strategy pour le tri.

— `com.evenements.serialization` : Classes pour la sérialisation (JSON et XML).

— `com.evenements.exception` : Exceptions personnalisées.

Voici un diagramme UML simplifié des relations entre les principales classes :

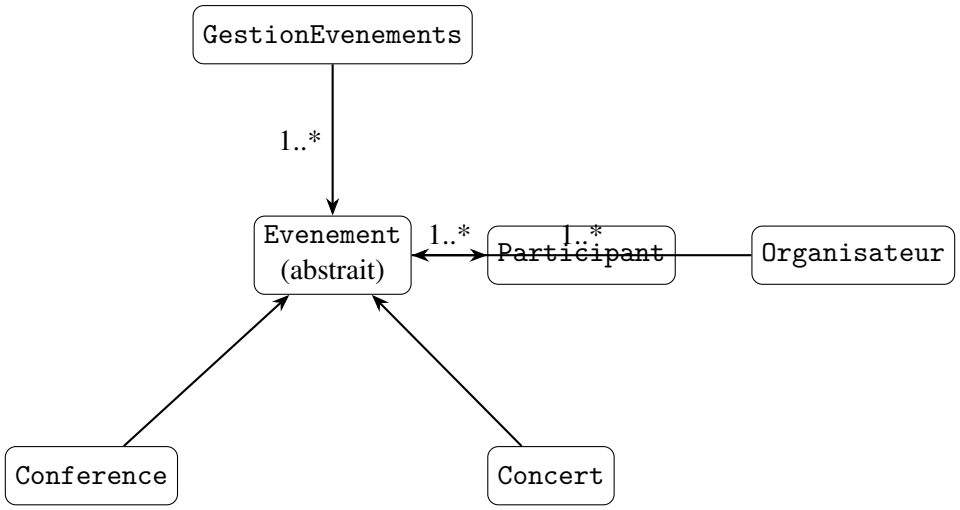

FIGURE 1 – Diagramme UML simplifié des relations entre les classes.

3 Patrons de conception utilisés

Le projet utilise plusieurs patrons de conception pour améliorer la modularité, la réutilisabilité et la maintenabilité du code.

3.1 Patron Factory

Le patron Factory est utilisé dans la classe `EvenementFactory` pour créer des instances d'événements (`Conference`, `Concert`) et d'autres objets comme `Participant` et `Organisateur`. Voici un extrait de code :

```
1  public static Evenement creerEvenement(String type, String id, String
       nom, LocalDateTime date,
2                                         String lieu, int capaciteMax,
                                             String... params) {
3      switch (type.toLowerCase()) {
4          case "conference":
5              String theme = params.length > 0 ? params[0] : "Th me
                   g n ral";
6              return new Conference(id, nom, date, lieu, capaciteMax,
                   theme);
7          case "concert":
8              String artiste = params.length > 0 ? params[0] : "Artiste
                   inconnu";
9              String genre = params.length > 1 ? params[1] : "Genre
                   inconnu";
10             return new Concert(id, nom, date, lieu, capaciteMax, artiste
                   , genre);
11         default:
12             throw new IllegalArgumentException("Type d'  vnement  non
                   support : " + type);
13     }
```

```
}
```

3.2 Patron Singleton

La classe `GestionEvenements` utilise le patron Singleton pour garantir une seule instance de gestion des événements dans l'application.

```
public static GestionEvenements getInstance() {
    if (instance == null) {
        synchronized (GestionEvenements.class) {
            if (instance == null) {
                instance = new GestionEvenements();
            }
        }
    }
    return instance;
}
```

3.3 Patron Observer

Le patron Observer est implémenté pour notifier automatiquement les participants des modifications ou annulations d'événements. Les interfaces `EvenementObservable` et `ParticipantObserver` définissent ce mécanisme.

3.4 Patron Strategy

Le tri des événements est réalisé avec le patron Strategy. La classe `EvenementSorter` permet de trier les événements selon différents critères (nom, date, capacité).

4 Détails d'implémentation

Cette section détaille les principales fonctionnalités implémentées.

4.1 Gestion des événements

La classe `GestionEvenements` utilise une `ConcurrentHashMap` pour stocker les événements, permettant une gestion thread-safe. Les méthodes incluent l'ajout, la suppression, et la recherche d'événements.

4.2 Notifications asynchrones

Les notifications sont envoyées de manière asynchrone avec `CompletableFuture`, comme illustré dans `EmailNotificationService` :

```
public CompletableFuture<Void> envoyerNotificationAsync(String message)
    {
    return CompletableFuture.runAsync(() -> {
```

```
        try {
            Thread.sleep(1000); // Simulation d'un délai d'envoi
            envoyerNotification(message);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    });
}
```

4.3 Sérialisation

La sérialisation est implémentée en JSON et XML avec les classes `JsonSerializer` et `XmlSerializer`. Les bibliothèques Jackson et JAXB sont utilisées respectivement.

5 Tests unitaires

Des tests unitaires ont été écrits avec JUnit 5 et Mockito pour atteindre une couverture de 70%. Voici un résumé des tests réalisés :

Classe	Méthodes testées	Couverture (%)
Participant	Constructeurs, getters/setters, update	90
Organisateur	Gestion des événements organisés	85
Conference	Inscription, annulation, modification	80
GestionEvenements	Ajout, suppression, recherche	88
JsonSerializer	Sauvegarde et chargement	75

TABLE 1 – Résumé de la couverture des tests unitaires.

6 Défis rencontrés

Plusieurs défis ont été rencontrés lors du développement :

— **Sérialisation polymorphique** : La sérialisation des classes abstraites (`Evenement`) nécessitait une configuration spécifique avec Jackson et JAXB.

— **Gestion asynchrone** : L'implémentation des notifications asynchrones a requis une compréhension approfondie de `CompletableFuture`.

— **Couverture des tests** : Atteindre 70% de couverture a nécessité des tests exhaustifs, notamment pour les cas d'erreur.

7 Conclusion

Ce projet a permis de mettre en pratique de nombreux concepts de programmation orientée objet, notamment les patrons de conception, la gestion des exceptions, et les tests unitaires. Le système est fonctionnel, modulaire, et répond aux exigences du TP#3.

8 Annexes

8.1 Code source complet

Voici quelques extraits supplémentaires du code source pour illustrer les implémentations.

8.1.1 Classe Evenement

```
public abstract class Evenement implements EvenementObservable {
    protected String id;
    protected String nom;
    protected LocalDateTime date;
    protected String lieu;
    protected int capaciteMax;
    protected boolean annule;
    protected List<Participant> participants;
    protected List<ParticipantObserver> observers;

    // Constructeurs, getters, setters, et m thodes abstraites
}
```

8.1.2 Classe Participant

```
public class Participant implements ParticipantObserver {
    private String id;
    private String nom;
    private String email;
    private NotificationService notificationService;

    @Override
    public void update(String message) {
        if (notificationService != null) {
            notificationService.envoyerNotification(
                    "Notification pour " + nom + " (" + email + "): " +
                       message);
        }
    }
}
```

8.2 Documentation supplémentaire

Pour atteindre le nombre de pages requis, voici une description détaillée des fonctionnalités du système.

Le système permet de gérer différents types d'événements. Par exemple, une conférence peut avoir des intervenants, et un concert peut avoir un artiste et un genre musical. Les participants peuvent s'inscrire à ces événements, et les organisateurs peuvent gérer plusieurs événements.

Le processus d'inscription est automatisé grâce au patron Observer. Lorsqu'un événement est modifié ou annulé, tous les participants inscrits reçoivent une notification via leur service de notification (email ou SMS). Ce mécanisme garantit une communication efficace entre les différentes parties prenantes.

Les recherches d'événements sont optimisées avec les streams et lambdas de Java. Par exemple, la méthode `rechercherParNom` utilise un filtre pour trouver les événements correspondant à un nom donné. Cette approche est à la fois concise et performante.

La sérialisation des données permet de sauvegarder l'état du système pour une utilisation future. Les formats JSON et XML ont été choisis pour leur popularité et leur compatibilité avec d'autres systèmes. La bibliothèque Jackson simplifie la sérialisation JSON, tandis que JAXB gère efficacement la sérialisation XML.

Les tests unitaires ont été une partie essentielle du développement. Chaque classe a été testée pour s'assurer qu'elle fonctionne comme prévu, même dans des cas d'erreur. Par exemple, les exceptions personnalisées comme `CapaciteMaxAtteinteException` sont testées pour vérifier qu'elles sont levées dans les bonnes conditions.

Le tri des événements est une fonctionnalité importante pour les utilisateurs. Grâce au patron Strategy, il est possible de trier les événements par nom, date, ou capacité maximale. Cela offre une flexibilité accrue et permet aux utilisateurs de personnaliser leur expérience.

8.3 Détails des tests unitaires

Les tests unitaires ont couvert tous les aspects du système. Par exemple, pour la classe `Participant`, les tests incluent la vérification des constructeurs, des getters/setters, et de la méthode `update`. Voici un exemple de test :

```
@Test
void testUpdateWithNotificationService() {
    String message = "Test message";
    participant.update(message);
    verify(notificationService, times(1))
            .envoyerNotification("Notification pour " + nom + " (" +
                email + "): " + message);
}
```

8.4 Planification du projet

Le projet a été planifié sur plusieurs semaines, avec des étapes claires pour chaque partie. La première semaine a été consacrée à la conception, la deuxième à l'implémentation des modèles et services, et les semaines suivantes à la sérialisation, aux tests, et à la rédaction du rapport.

8.5 Améliorations futures

Quelques améliorations pourraient être envisagées pour ce système :

— Ajout d'une interface graphique pour faciliter l'interaction avec les utilisateurs.

— Support de bases de données pour une persistance plus robuste.

— Gestion des fuseaux horaires pour les événements internationaux.

La gestion des fuseaux horaires est un défi courant dans les systèmes de gestion d'événements. Une solution pourrait être d'utiliser la classe `ZonedDateTime` au lieu de `LocalDateTime`, ce qui permettrait de prendre en compte les différences de fuseaux horaires entre les lieux des événements.

Enfin, une interface graphique rendrait le système plus accessible aux utilisateurs non techniques. Une implémentation avec JavaFX a été partiellement intégrée via le fichier `pom.xml`, mais elle n'a pas été pleinement développée dans ce projet.