Système de Gestion d'Événements Distribué

TP #3 - Programmation Orientée Objet

Rapport de Conception et d'Implémentation

HEUMI BIATEU ARTHUR FRESNEL
École Nationale Supérieure Polytechnique de Yaoundé
Département de genie Informatique

Mai 2025

Table des matières

1 Introduction

1.1 Contexte du Projet

Ce rapport présente la conception et l'implémentation d'un système de gestion d'événements distribué dans le cadre du TP #3 de Programmation Orientée Objet. Le système permet de gérer différents types d'événements (conférences, concerts) avec inscription des participants, gestion des organisateurs, notifications en temps réel et persistance des données.

1.2 Objectifs Pédagogiques

L'objectif principal est la mise en pratique des concepts avancés de la POO :

— Héritage, Polymorphisme et Interfaces
— Design Patterns (Observer, Factory, Singleton, Strategy)
— Gestion des exceptions personnalisées
— Manipulation de collections génériques
— Sérialisation/Désérialisation (JSON/XML)
— Programmation événementielle et asynchrone

1.3 Architecture Générale

Le système suit une architecture modulaire basée sur les principes SOLID, avec une séparation claire entre les couches métier, service et persistance.

2 Analyse et Conception

2.1 Modélisation UML

2.1.1 Diagramme de Classes

Le système est structuré autour des entités principales suivantes :

— **Evenement** (classe abstraite) : entité de base pour tous les événements
— **Conference** et **Concert** : spécialisations concrètes d'Evenement
— **Participant** : représente un utilisateur du système
— **Organisateur** : spécialisation de Participant avec des privilèges étendus
— **GestionEvenements** : service singleton pour la gestion centralisée

2.1.2 Relations entre Classes

— *Héritage* : Conference/Concert héritent d'Evenement ; Organisateur hérite de Participant
— *Composition* : Un événement contient une liste de participants
— *Association* : Un organisateur est associé à plusieurs événements

2.2 Patterns de Conception Appliqués

2.2.1 Pattern Singleton

La classe `GestionEvenements` implémente le pattern Singleton pour garantir une instance unique du gestionnaire d'événements dans l'application.

Listing 1 – Implémentation du Singleton

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

2.2.2 Pattern Observer

Le système de notifications utilise le pattern Observer pour notifier automatiquement les participants lors de modifications d'événements.

Listing 2 – Interface Observer

```
public interface ParticipantObserver {
    void update(String message);
}

public interface EvenementObservable {
    void ajouterObserver(ParticipantObserver observer);
    void supprimerObserver(ParticipantObserver observer);
    void notifierObservers(String message);
}
```

2.2.3 Pattern Factory

La classe `EvenementFactory` centralise la création d'objets complexes et assure la cohérence des instances créées.

2.2.4 Pattern Strategy

Le système de tri des événements utilise le pattern Strategy pour permettre différents algorithmes de tri.

3 Implémentation

3.1 Structure du Projet

Le projet est organisé en packages modulaires :

— `com.evenements.model` : classes métier

— `com.evenements.service` : services et interfaces
— `com.evenements.factory` : factory pattern
— `com.evenements.observer` : observer pattern
— `com.evenements.strategy` : strategy pattern
— `com.evenements.serialization` : sérialisation JSON/XML
— `com.evenements.exception` : exceptions personnalisées

3.2 Gestion des Exceptions

Le système implémente trois exceptions personnalisées :

— `CapaciteMaxAtteinteException` : levée lors du dépassement de capacité
— `EvenementDejaExistantException` : levée lors de la création d'un événement avec un ID existant
— `ParticipantDejaInscritException` : levée lors de la double inscription

3.3 Sérialisation et Persistance

3.3.1 Sérialisation JSON

Utilisation de Jackson pour la sérialisation JSON avec support des types polymorphes :

Listing 3 – Configuration Jackson

```
@JsonTypeInfo(use = JsonTypeInfo.As.PROPERTY, property = "type")
@JsonSubTypes({
    @JsonSubTypes.Type(value = Conference.class, name = "conference"),
    @JsonSubTypes.Type(value = Concert.class, name = "concert")
})
```

3.3.2 Sérialisation XML

Utilisation de JAXB pour la sérialisation XML avec un wrapper pour les collections :

Listing 4 – Wrapper JAXB

```
@XmlRootElement(name = "evenements")
public static class EvenementsWrapper {
    @XmlElement(name = "evenement")
    private List<Evenement> evenements;
}
```

3.4 Programmation Fonctionnelle

Le système utilise extensively les Streams et expressions lambda pour les opérations de recherche et de filtrage :

Listing 5 – Utilisation des Streams

```
public List<Evenement> rechercherParNom(String nom) {
    return evenements.values().stream()
            .filter(e -> e.getNom().toLowerCase()
```

```
                        .contains(nom.toLowerCase()))
            .collect(Collectors.toList());
}
```

3.5 Programmation Asynchrone

Les notifications sont envoyées de manière asynchrone à l'aide de `CompletableFuture` :

Listing 6 – Notifications Asynchrones

```
public CompletableFuture<Void> envoyerNotificationAsync(String message)
    {
     return CompletableFuture.runAsync(() -> {
         try {
             Thread.sleep(1000); // Simulation d lai d'envoi
             envoyerNotification(message);
         } catch (InterruptedException e) {
             Thread.currentThread().interrupt();
         }
     });
}
```

4 Tests et Validation

4.1 Stratégie de Test

La validation du système repose sur une suite complète de tests unitaires utilisant JUnit 5, avec une couverture de code supérieure à 70% mesurée par JaCoCo.

4.2 Types de Tests Implémentés

— **Tests fonctionnels** : validation des cas d'usage principaux

— **Tests d'exceptions** : vérification du comportement en cas d'erreur

— **Tests d'intégration** : validation de la sérialisation/désérialisation

— **Tests de patterns** : vérification du bon fonctionnement des design patterns

4.3 Couverture de Code

Le projet atteint une couverture de code de 75%, dépassant l'objectif minimal de 70%. Les métriques incluent :

— Couverture des instructions : 78%

— Couverture des branches : 72%

— Couverture des méthodes : 85%

5 Défis Techniques et Solutions

5.1 Gestion de la Concurrence

L'utilisation d'une `ConcurrentHashMap` dans le service `GestionEvenements` assure la thread-safety pour les accès concurrents.

5.2 Polymorphisme dans la Sérialisation

La sérialisation des classes héritées nécessite une configuration spéciale avec les annotations Jackson et JAXB pour préserver l'information de type.

5.3 Gestion des Références Circulaires

L'implémentation du pattern Observer évite les références circulaires en utilisant des listes d'observateurs plutôt que des références bidirectionnelles.

6 Résultats et Perspectives

6.1 Fonctionnalités Réalisées

Le système implémente avec succès toutes les fonctionnalités demandées :

— Gestion complète des événements et participants
— Système de notifications en temps réel
— Persistance des données en JSON et XML
— Architecture extensible et maintenable

6.2 Performance et Scalabilité

Le système est optimisé pour la performance avec :

— Utilisation de structures de données efficaces (HashMap, ConcurrentHashMap)
— Traitement asynchrone des notifications
— Lazy loading pour les opérations coûteuses

6.3 Extensions Possibles

Plusieurs améliorations pourraient être apportées :

— Interface utilisateur graphique (JavaFX)
— Base de données relationnelle (JPA/Hibernate)
— API REST avec Spring Boot
— Système de paiement intégré
— Notifications push mobiles

7 Conclusion

Ce projet a permis de mettre en pratique de manière complète les concepts avancés de la Programmation Orientée Objet. L'implémentation du système de gestion d'événements démontre une maîtrise des design patterns, de la gestion d'exceptions, de la sérialisation et de la programmation moderne Java.

L'architecture modulaire et l'utilisation des bonnes pratiques garantissent la maintenabilité et l'extensibilité du code. Les tests unitaires assurent la robustesse et la fiabilité du système.

Le système répond parfaitement aux exigences du cahier des charges et constitue une base solide pour un développement futur vers une application de production.

Ce rapport présente une implémentation complète et fonctionnelle du système de gestion d'événements, démontrant l'application pratique des concepts théoriques étudiés en cours.