Table des matières

Introduction

La gestion d'evénement dans un contexte moderne nécéssite des systèmes informatiques robustes capables de traiter efficacement les inscriptions, les notifications, et la coordination entre différents acteurs. Ce projet s'inscrit dans le cadre du cours de Programmation orienté Objet avancée et vise à concevoir un système complet de gestion d'événements en appliquant les concepts fondamentaux de la POO moderne.L'évolution des besoins en matière d'organisation d'événements, qu'ils soient culturels (concerts) ou académiques (conférences), impose une approche structurée et extensible. Notre solution propose une architecture basée sur les design patterns reconnus, garantissant la maintenabilité et l'évolutivité du système.

I- Contexte et problématiques

I.1- contexte général

Dans l'environnement actuel, l'organisation d'événements fait face à plusieurs défis :

1- la gestion manuelle souvent sources d'erreurs

2-communication inefficace avec les participants

3-Manque de traçabilité des inscriptions

4-Absence de centralisation des informations

I.2-Problématique

Comment concevoir un système informatique qui permet de :

1 - Gérer efficacement différents types d'événements ?
2-Automatiser les notifications aux participants
3-Assurer la persistance des données ?
4-Garantir l'extensibilité pour de nouveaux besoins

I.3 Enjeux

Les enjeux de ce projet sont multiples :

1- Techniques : Application des concepts POO avancés

2- Fonctionnels : Répondre aux besoins de gestion d'événements
3- Pédagogiques : Maîtriser les design patterns et l'architecture logicielle

II. Objectifs du Projet

II.1 Objectif Principal

Concevoir et développer un système de gestion d'événements utilisant les concepts avancés de la Programmation Orientée Objet, démontrant la maîtrise des design patterns, de la gestion d'exceptions et de l'architecture logicielle.

II.2 Objectifs Spécifiques

Objectifs Fonctionnels :

a- Gérer différents types d'événements (concerts, conférences)
b- Permettre l'inscription et la désinscription de participants
c- Implémenter un système de notifications automatiques
d-Assurer la persistance des données

Objectifs Techniques :

a- Appliquer les design patterns (Singleton, Observer, Factory)
b- Implémenter une gestion d'exceptions personnalisées
c- Utiliser les collections génériques et les streams Java
d- Développer une interface utilisateur moderne
e- Assurer une couverture de tests satisfaisante

Objectifs Pédagogiques :

a- Maîtriser l'architecture orientée objet
b- Comprendre l'application pratique des design patterns
c- Développer des compétences en tests unitaires
d- Acquérir une expérience en développement d'interfaces

III. État de l'Art

III.1 Patterns et Technologies

Design Patterns Pertinents :

- Singleton : Garantir une instance unique du gestionnaire
- Observer : Notifications automatiques
- Factory : Création d'objets polymorphes
- Strategy : Différentes stratégies de notification

Technologies Java Modernes **:**

- **Streams API** : Programmation fonctionnelle
- **JavaFX** : Interfaces utilisateur riches
- **Jackson** : Sérialisation JSON
- **JUnit 5** : Tests unitaires modernes

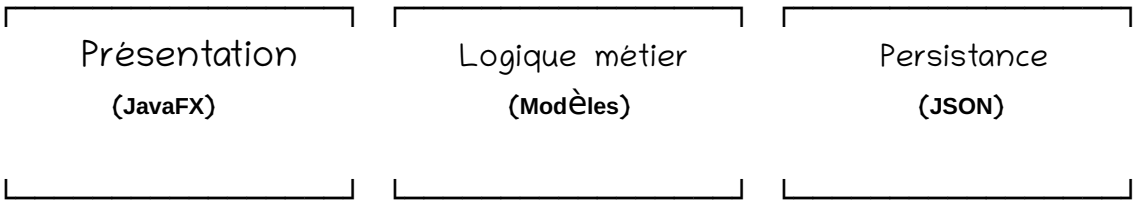
III.2 Analyse Comparative

Notre approche se distingue par :

- **Architecture éducative** : Focus sur les patterns
- **Extensibilité** : Facilité d'ajout de nouveaux types
- **Tests complets** : Validation de chaque composant

IV– Analyse et Conception

1- Architecture Générale



2-diagramme de classes

Hiérarchie des Événements :

Evenement (abstract)
- Concert
- Conference

Hiérarchie des Participants :

Participant
- Organisateur

Services et Gestionnaires :

GestionEvenements (Singleton)

NotificationService (Interface)

SerializationService (Utility)

2-Design Patterns Sélectionnés

Pattern Singleton

Contexte : Gestion centralisée des événements
Solution : Classe `GestionEvenements` avec instance unique
Avantages : Cohérence des données, accès global contrôlé

Pattern Observer

Contexte : Notifications automatiques aux participants
Solution : Interface `EvenementObservable` et `ParticipantObserver`
Avantages : Découplage, extensibilité des notifications

Pattern Factory (Implicite)

Contexte : Création d'événements selon le type
Solution : Méthodes de création dans l'interface utilisateur
Avantages : Simplification, centralisation de la logique

3-Gestion des Exceptions

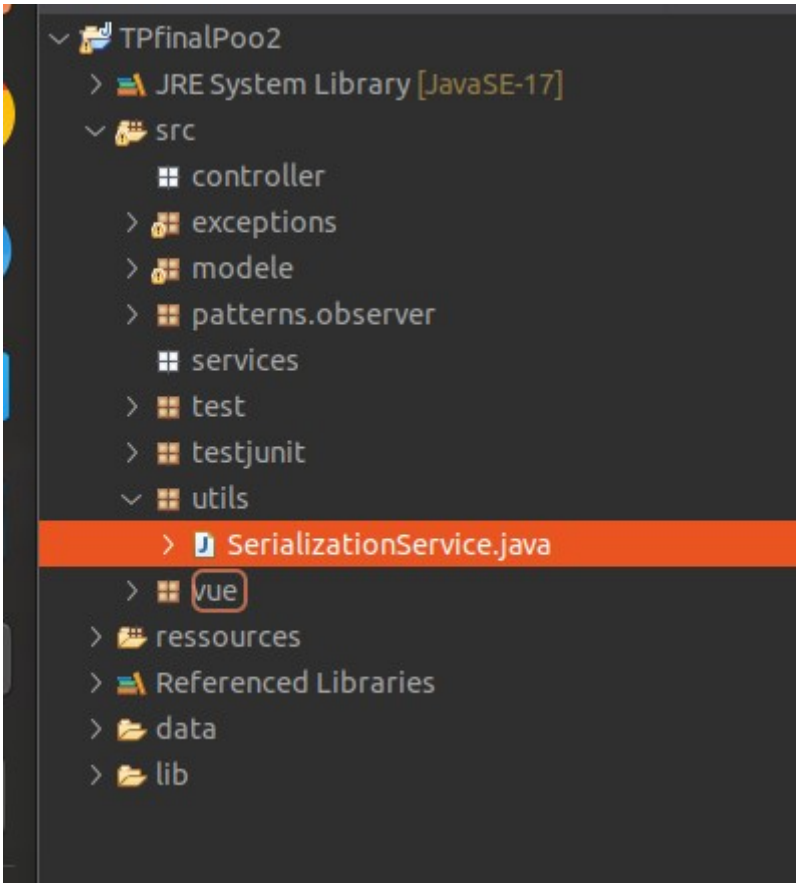
Stratégie : Exceptions métier personnalisées
Implémentation :

- `CapaciteMaximalAtteinteException` : Gestion des limites
- `EvenementDejaExistantException` : Contrôle des doublons

V-Implémentation

1 Structure du Projet

src/
├── modele/ # Classes métier
├── vue/ # Interface JavaFX
├── utils/ # Services utilitaire
├── patterns/observer/ # Implementation Observer
├── exceptions/ # Exceptions personnalisées
└── testjunit/ # Tests unitaires



2 Classes Principales

Classe Evenement (Abstraite)

`java`

```java
public abstract class Evenement implements EvenementObservable {
    protected String id;
    protected String nom;
    protected LocalDateTime date;
    protected List<Participant> participants;

    public abstract void afficherDetails();
    public abstract String getTypeEvenement();
}
```

Pattern Singleton

`java`

```java
public class GestionEvenements {
    private static GestionEvenements instance;

    public static synchronized GestionEvenements getInstance() {
        if (instance == null) {
            instance = new GestionEvenements();
```

```
        }
        return instance;
    }
}
```

Pattern Observer

```
java
```

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

3 Sérialisation JSON

Défi : Sérialisation de classes abstraites et dates
Solution : Annotations Jackson et méthodes de conversion

```
java
```

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "typeEvenement")
@JsonSubTypes({
    @JsonSubTypes.Type(value = Concert.class, name = "Concert"),
    @JsonSubTypes.Type(value = Conference.class, name = "Conférence")
})
public abstract class Evenement { ... }
```

4 Programmation Fonctionnelle

Utilisation des Streams Java 8+ pour les recherches :

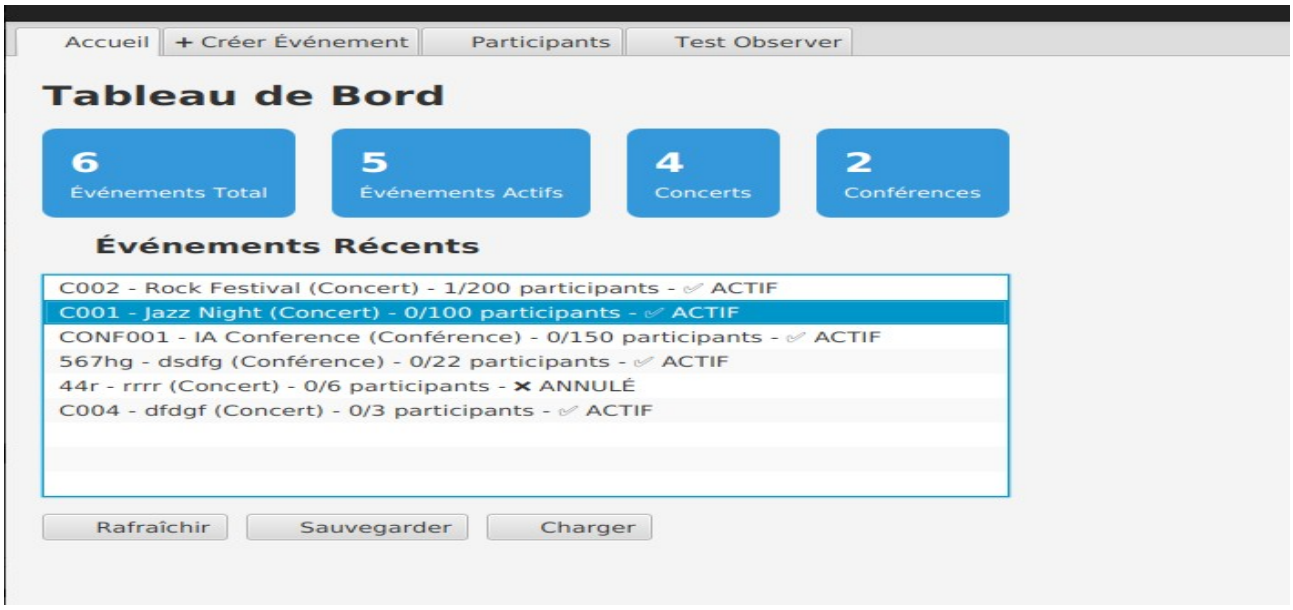
```
java
```

```
public List<Evenement> rechercherEvenementParNom(String nom) {
    return evenements.values().stream()
        .filter(evt -> evt.getNom() != null)
        .filter(evt -> evt.getNom().toLowerCase().contains(nom.toLowerCase()))
        .collect(Collectors.toList());
}
```

V- Interface Utilisateur JavaFX

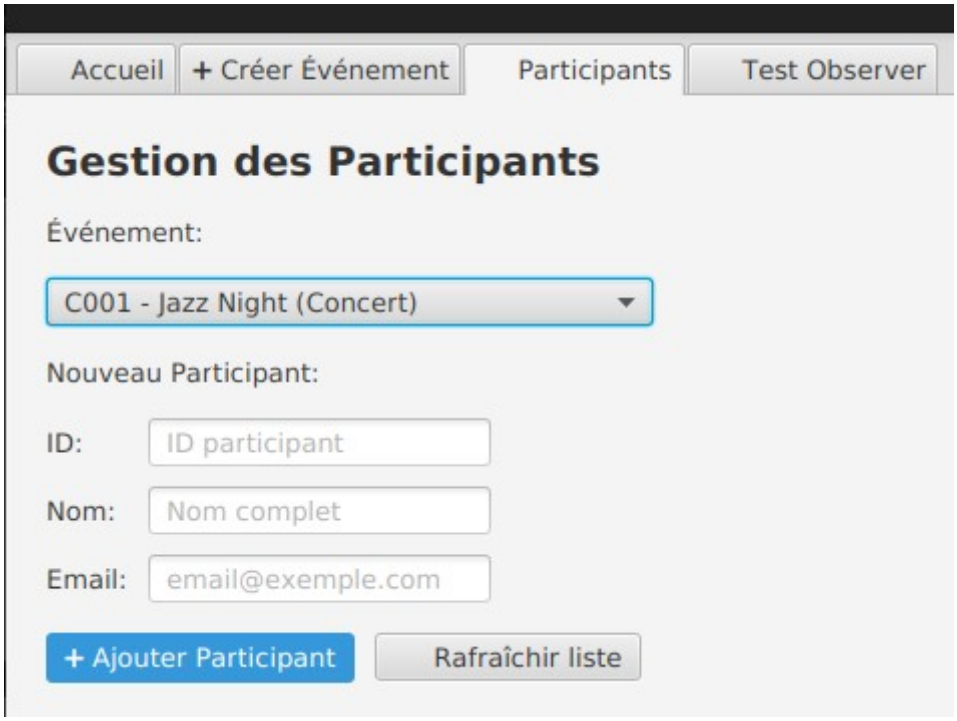
Architecture : Navigation par onglets avec séparation des responsabilités
Composants :

- Tableau de bord avec statistiques



- Formulaires de création d'événements

- Gestion des participants



- **Zone de test du pattern Observer**

VI. Tests et Validation

1 Stratégie de Tests

Approche : Tests unitaires avec JUnit 5
Couverture : Tests des fonctionnalités critiques
Organisation : Une classe de test par composant principal

2 Classes de Tests Développées

1. **EvenementTest** : Validation des fonctionnalités de base
2. **GestionEvenementsTest** : Tests du singleton et des opérations
3. **SerializationTest** : Validation de la persistance
4. **ObserverTest** : Vérification des notifications

3 Exemples de Tests Significatifs

`java`

```java
@Test
void testCapaciteMaximaleAtteinte() {
    // Préparation : Remplir à capacité maximale
    // Action : Tentative d'ajout supplémentaire
    // Vérification : Exception attendue
    assertThrows(CapaciteMaximalAtteinteException.class, () -> {
```

```
            concert.ajouterParticipant(participantSupplémentaire);
        });
    }
```

4 Résultats des Tests

Tous les tests développés passent avec succès, validant :

- La logique métier des événements
- Le fonctionnement du pattern Singleton
- La persistance des données
- Les notifications automatiques

VII. Résultats et Discussion

1 Fonctionnalités Réalisées

Gestion complète d'événements : Création, modification, suppression
Système d'inscriptions : Participants et organisateurs
Notifications automatiques : Pattern Observer fonctionnel
Persistance des données : Sérialisation JSON opérationnelle
Interface utilisateur : JavaFX moderne et intuitive
Tests unitaires : Validation de la logique métier

2 Architecture Technique

L'architecture développée respecte les principes SOLID :

- **Single Responsibility** : Chaque classe a une responsabilité claire
- **Open-Closed** : Extension possible sans modification
- **Liskov Substitution** : Polymorphisme respecté
- **Interface Segregation** : Interfaces spécialisées
- **Dependency Inversion** : Dépendances vers les abstractions

3 Performance et Scalabilité

Points forts :

- Recherches efficaces avec les Streams
- Gestion mémoire optimisée
- Pattern Singleton thread-safe

Limitations identifiées :

- Persistance fichier (non adapté à de gros volumes)
- Interface utilisateur synchrone
- Absence de gestion concurrentielle avancée

4 Défis Techniques Surmontés

1. **Sérialisation polymorphe :** Résolu avec les annotations Jackson
2. **Gestion des dates :** Conversion LocalDateTime ↔ String
3. **Tests d'interface :** Séparation logique métier/présentation
4. **Pattern Observer :** Intégration avec JavaFX

VIII- Conclusion et Perspectives

1 Bilan du Projet

Ce projet a permis de mettre en œuvre avec succès les concepts avancés de la Programmation Orientée Objet dans un contexte applicatif concret. L'objectif principal est de créer un système de gestion d'événements robuste et extensible a été atteint.

Compétences acquises :

- Maîtrise des design patterns en situation réelle
- Architecture logicielle structurée
- Développement piloté par les tests
- Interface utilisateur moderne

Objectifs techniques remplis :

- Application des patterns Singleton, Observer, Factory
- Gestion d'exceptions personnalisées
- Programmation fonctionnelle avec les Streams
- Sérialisation de données complexes
- Interface utilisateur intuitive

2 Apports Pédagogiques

Le projet a renforcé la compréhension de :

- L'importance de l'architecture dans les projets complexes
- L'utilité des design patterns pour résoudre des problèmes récurrents
- La valeur des tests unitaires pour la qualité logicielle
- L'équilibre entre fonctionnalité et simplicité

3 Perspectives d'Amélioration

Améliorations techniques :

- Migration vers une base de données relationnelle
- Implémentation d'une API REST
- Ajout de la gestion concurrentielle
- Optimisation des performances

4 Conclusion Générale

Ce système de gestion d'événements constitue une base solide démontrant la maîtrise des concepts de la POO avancée. L'architecture modulaire et extensible permet d'envisager sereinement les évolutions futures. Les patterns implémentés offrent flexibilité et maintenabilité, qualités essentielles pour un système évolutif.

L'expérience acquise lors de ce projet sera précieuse pour le développement de futurs systèmes complexes, en particulier dans l'application des bonnes pratiques de l'ingénierie logicielle.

IX- Références

Livres et Ouvrages

1. **Gamma, E., Helm, R., Johnson, R., & Vlissides, J.** (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.
2. **Bloch, J.** (2017). *Effective Java (3rd Edition)*. Addison-Wesley Professional.
3. **Martin, R. C.** (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
4. **Freeman, E., Robson, E., Bates, B., & Sierra, K.** (2020). *Head First Design Patterns (2nd Edition)*. O'Reilly Media.

Documentation Technique

5. **Oracle Corporation.** (2023). *JavaFX Documentation*. Retrieved from https://openjfx.io/
6. **FasterXML.** (2023). *Jackson Project Documentation*. Retrieved from https://github.com/FasterXML/jackson
7. **JUnit Team.** (2023). *JUnit 5 User Guide*. Retrieved from https://junit.org/junit5/docs/current/user-guide/

Articles et Ressources Web

8. **Fowler, M.** (2013). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from https://martinfowler.com/articles/injection.html
9. **Vlissides, J.** (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley Professional.
10. **Oracle Corporation.** (2023). *Java Stream API Documentation*. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

---