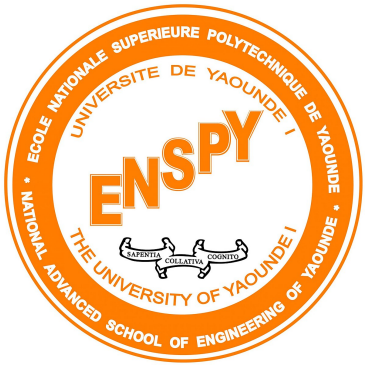
RAPPORT TP FINAL POO2

Système de Gestion d'Événements



Analyse Complète et Documentation Technique

Rédigé et Présenté par :

PANGUI PIANNE PEGUY

Sous la supervision de :

Dr KUNGNE

Département de Génie Logicielle

26 mai 2025

Table des matières

1 Introduction

Le système de gestion d'événements est une application Java conçue pour gérer des événements tels que des concerts et des conférences, en permettant l'ajout, la suppression, la recherche, la sauvegarde et le chargement d'événements, ainsi que la gestion des participants et des notifications. L'application utilise des principes de programmation orientée objet (POO), des patrons de conception, et des technologies modernes pour assurer modularité, extensibilité et robustesse.

Objectif du rapport

Ce rapport est structuré comme suit :

- **Choix des outils** : Explication des technologies et bibliothèques utilisées
- **Choix de conception** : Analyse des patrons de conception et des principes appliqués
- **Diagrammes UML** : Représentations graphiques des classes, interactions et comportements
- **Recommandations** : Suggestions d'améliorations pour le système
- **Conclusion** : Résumé des points clés

2 Choix des Outils

2.1 Langage de Programmation : Java

Raisons du choix :

- **Portabilité** : Java est indépendant de la plateforme grâce à la JVM (Java Virtual Machine), permettant au système de fonctionner sur différents systèmes d'exploitation sans modification.
- **Écosystème riche** : Java dispose de nombreuses bibliothèques et frameworks (par exemple, Jackson pour la sérialisation JSON), ce qui accélère le développement.
- **Typage fort** : Le typage statique réduit les erreurs à l'exécution, essentiel pour un système de gestion robuste.
- **Communauté et support** : Une large communauté et une abondance de documentation facilitent la résolution de problèmes.

Impact : Java est idéal pour un système nécessitant une structure orientée objet claire, une gestion des exceptions robuste, et une compatibilité avec des formats de données comme JSON.

2.2 Bibliothèque Jackson

Utilisation : Jackson est utilisé pour la sérialisation et la désérialisation des objets Java en JSON (et vice-versa) dans les méthodes `sauvegarderEvenements`, `chargerEvenements`, `sauvegarderParticipants`, et `chargerParticipants`.

Raisons du choix :

— **Performance** : Jackson est rapide et optimisé pour la manipulation de grandes quantités de données.

— **Support des types complexes** : Avec le module `JavaTimeModule`, Jackson gère les types modernes de Java comme `LocalDateTime`.

— **Facilité d'intégration** : Les annotations et l'API de Jackson simplifient la sérialisation des objets complexes.

Impact : Permet une persistance des données dans des fichiers JSON, offrant une solution légère par rapport à une base de données relationnelle pour un système de petite à moyenne échelle.

2.3 CompletableFuture pour les Notifications

Utilisation : L'interface `NotificationService` utilise `CompletableFuture<Void>` pour envoyer des notifications de manière asynchrone.

Raisons du choix :

— **Asynchronisme** : Les notifications (par exemple, par email ou SMS) peuvent être longues, et `CompletableFuture` permet de ne pas bloquer le thread principal.

— **Programmation réactive** : Favorise une architecture non bloquante, améliorant la scalabilité.

— **Flexibilité** : Permet d'implémenter différents services de notification (email, SMS, etc.) sans modifier le code existant.

Impact : Améliore les performances en gérant les opérations lentes de manière asynchrone, tout en offrant une interface extensible.

2.4 Collections Java (HashMap, ArrayList)

Utilisation :

— `HashMap` dans `GestionEvenements` pour stocker les événements avec leur ID comme clé.

— `ArrayList` dans `Evenement` pour gérer les listes de participants et d'observateurs.

Raisons du choix :

— **Efficacité** : `HashMap` offre un accès en O(1) pour les recherches par ID, tandis qu'`ArrayList` est adapté pour des listes dynamiques de taille modérée.

— **Simplicité** : Les collections standard de Java sont bien documentées et faciles à utiliser.

— **Type-safety** : Les génériques assurent la sécurité des types lors de la manipulation des collections.

Impact : Fournit une gestion efficace et sûre des données en mémoire, adaptée à un système de gestion d'événements.

2.5 Java 8+ (Stream API)

Utilisation : La méthode `rechercherEvenement` dans `GestionEvenements` utilise les streams pour filtrer les événements par nom.

Raisons du choix :

— **Lisibilité** : Les streams permettent d'écrire un code concis et expressif.

— **Fonctionnalité** : Les opérations comme `filter` et `findFirst` simplifient les recherches dans les collections.

— **Performance** : Les streams peuvent être parallélisés si nécessaire pour les grandes collections.

Impact : Améliore la maintenabilité du code et prépare le système à une éventuelle parallélisation.

3 Choix de Conception

3.1 Architecture Orientée Objet

Principe : Le système utilise une hiérarchie de classes avec une classe abstraite `Evenement` comme base pour `Concert` et `Conference`.

Raisons du choix :

— **Héritage** : Permet de partager les attributs et comportements communs (par exemple, `id`, `nom`, `participants`) tout en spécialisant les sous-classes.

— **Polymorphisme** : La méthode abstraite `afficherDetails` est redéfinie dans chaque sous-classe pour fournir des détails spécifiques.

— **Encapsulation** : Les attributs sont privés avec des getters/setters, protégeant l'intégrité des données.

Principe SOLID

Cette approche facilite l'ajout de nouveaux types d'événements (par exemple, `Festival`) sans modifier le code existant, respectant le principe **Open/Closed** de SOLID.

3.2 Patron Singleton (GestionEvenements)

Implémentation : La classe `GestionEvenements` utilise le patron Singleton pour garantir une seule instance de gestion des événements.

Raisons du choix :

— **Contrôle d'accès** : Une seule instance centralise la gestion des événements, évitant les incohérences.

— **Ressources partagées** : Évite la duplication de la `HashMap` d'événements et de l'`ObjectMapper`.

Limitations

Le Singleton peut poser des problèmes dans des environnements multi-threads ou pour les tests unitaires. Le Singleton est implémenté de manière thread-safe avec une initialisation paresseuse (*lazy initialization*).

Impact : Simplifie l'accès global à la gestion des événements, mais il faut être prudent avec les tests et la scalabilité.

3.3 Patron Factory (Proposition d'Amélioration)

Observation : Actuellement, les objets `Concert` et `Conference` sont créés directement dans le code client.

Proposition : Introduire un **Factory Pattern** pour centraliser la création d'événements.

```
public class EvenementFactory {
    public static Evenement createEvenement(String type, String id,
            String nom, LocalDateTime date, String lieu,
            int capaciteMax, Map<String, Object> params) {
        switch (type) {
            case "Concert":
                return new Concert(id, nom, date, lieu, capaciteMax,
                    (String) params.get("artiste"),
                    (String) params.get("genreMusical"));
            case "Conference":
                return new Conference(id, nom, date, lieu, capaciteMax,
                    (String) params.get("theme"),
                    (List<String>) params.get("intervenants"));
            default:
                throw new IllegalArgumentException(
                    "Type d'  vnement  inconnu : " + type);
        }
    }
}
```

Listing 1 – Exemple d'implémentation Factory Pattern

Avantage : Simplifie l'instanciation et permet d'ajouter de nouveaux types d'événements sans modifier le code client.

3.4 Patron Observer

Implémentation : L'interface `EvenementObservable` et la classe `Participant` implémentent le patron Observer pour gérer les notifications aux participants.

Raisons du choix :

— **Dynamisme** : Permet d'ajouter ou de supprimer des observateurs (participants) à la volée.

— **Découplage** : Les événements ne dépendent pas de la manière dont les notifications sont envoyées.

— **Extensibilité** : Les observateurs peuvent être des participants, des organisateurs, ou même des services externes.

Impact : Assure une communication efficace entre les participants et les événements, avec une gestion centralisée des notifications.

3.5 Gestion des Exceptions

Implémentation : Des exceptions personnalisées comme `EvenementDejaExistantException` et `CapaciteMaxAtteinteException` sont utilisées pour gérer les erreurs spécifiques.

Raisons du choix :

— **Clarté** : Les exceptions personnalisées fournissent des messages d'erreur significatifs.

— **Robustesse** : Permet de gérer proprement les erreurs comme l'ajout d'un événement existant ou le dépassement de la capacité maximale.

Impact : Améliore la fiabilité du système en prévenant des erreurs prévisibles.

3.6 Persistance JSON

Implémentation : Les événements et les participants sont sauvegardés dans des fichiers JSON à l'aide de Jackson.

Raisons du choix :

— **Simplicité** : JSON est léger et lisible, adapté pour un système sans base de données.

— **Portabilité** : Les fichiers JSON peuvent être facilement partagés ou archivés.

Limitation

Pas adapté pour de grandes quantités de données ou des accès concurrents.

Impact : Fournit une solution de persistance simple, mais une base de données (par exemple, PostgreSQL) pourrait être envisagée pour une scalabilité accrue.

4 Diagrammes UML

4.1 Diagramme de Classes

Le diagramme de classes illustre la structure statique du système, y compris les relations d'héritage, d'association et de dépendance.

Explication :

— **Héritage** : `Concert` et `Conference` héritent de `Evenement`.

— **Interfaces** : `EvenementObservable` et `ParticipantObserver` implémentent le patron Observer.

— **Associations** :

  — `Evenement` contient une liste de `Participant` et de `ParticipantObserver`.

  — `GestionEvenements` gère une collection d'`Evenement`.

  — `Organisateur` est associé à une liste d'`Evenement`.

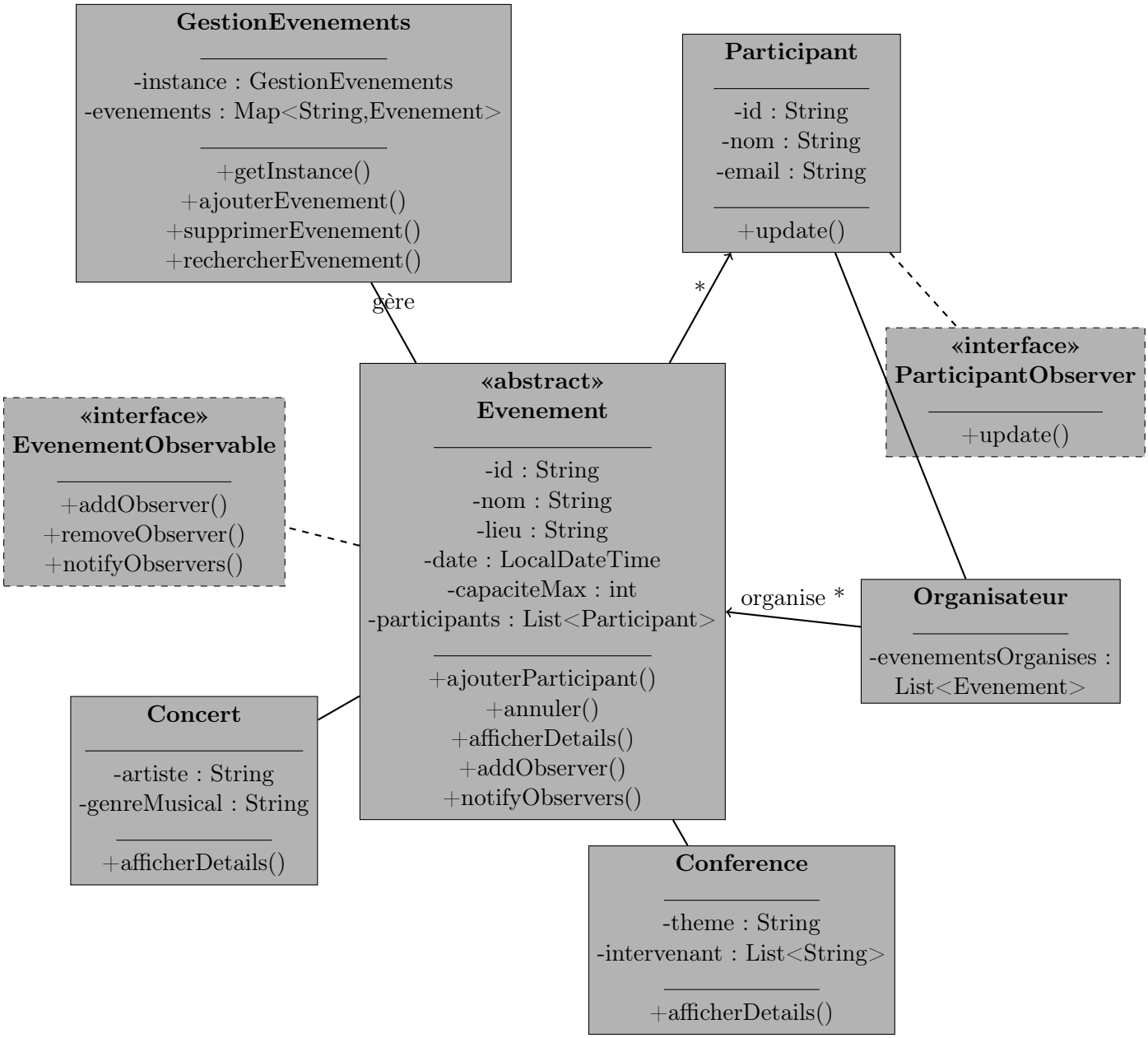


FIGURE 1 – Diagramme de classes du système de gestion d'événements

4.2 Diagramme de Cas d'Utilisation

Explication :

— **Utilisateur** : Gère les opérations CRUD sur les événements.

— **Organisateur** : Planifie et annule des événements.

— **Participant** : S'inscrit et reçoit des notifications.

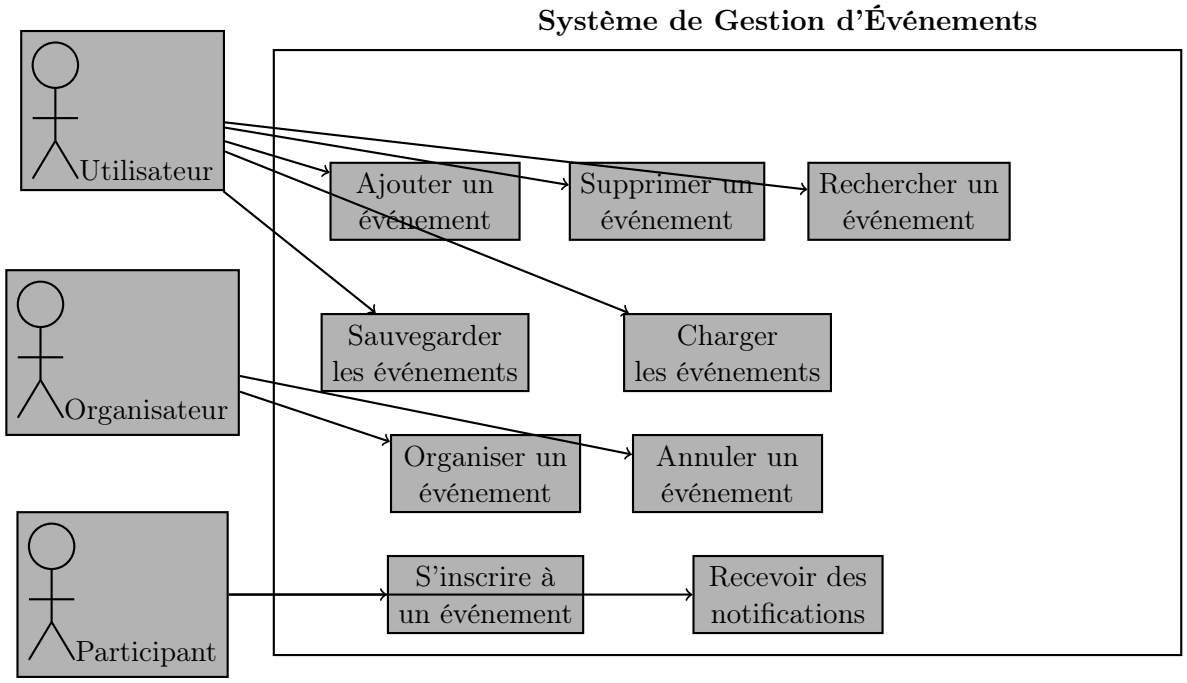


FIGURE 2 – Diagramme de cas d'utilisation

5 Conclusion

Le système de gestion d'événements est une application Java bien structurée, utilisant des principes de POO, des patrons de conception comme Singleton et Observer, et des technologies modernes comme Jackson et CompletableFuture. Les choix d'outils (Java, Jackson, collections Java) et de conception (héritage, encapsulation, asynchronisme) rendent le système robuste, extensible et adapté à la gestion d'événements de petite à moyenne échelle.

Points forts du système

- Architecture orientée objet claire et extensible
- Utilisation appropriée des patrons de conception
- Gestion asynchrone des notifications
- Persistance simple et efficace en JSON
- Code bien structuré et maintenable

Les diagrammes UML fournissent une vue claire de la structure et du comportement du système, tandis que les recommandations d'amélioration (base de données, synchronisation, tests) préparent le système à une évolution future. Avec quelques ajustements, ce système peut devenir une solution professionnelle pour la gestion d'événements à grande échelle.

6 Annexes

6.1 Code Source

Voir dépôt GitHub https://github.com/PanguiPeguy/Gestionnaire-d-Evenements

6.2 Matrice de Traçabilité des Exigences

TABLE 1 – Matrice de traçabilité des exigences fonctionnelles

Exigence	**Classe Responsable**	**Méthode**	**Status**
Ajouter un événement	GestionEvenements	ajouterEvenement()	Implémenté
Supprimer un événement	GestionEvenements	supprimerEvenement()	Implémenté
Rechercher un événement	GestionEvenements	rechercherEvenement()	Implémenté
Ajouter un participant	Evenement	ajouterParticipant()	Implémenté
Notifier les participants	Evenement	notifyObservers()	Implémenté
Sauvegarder les données	GestionEvenements	sauvegarderEvenements()	Implémenté
Charger les données	GestionEvenements	chargerEvenements()	Implémenté
Gérer la capacité max	Evenement	ajouterParticipant()	Implémenté
Gérer l'unicite des Evenement	Evenement	ajouterEvenement()	Implémenté

6.3 Glossaire Technique

API (Application Programming Interface) Interface de programmation d'application permettant la communication entre différents composants logiciels.

CompletableFuture Classe Java permettant la programmation asynchrone et non-bloquante.

Design Pattern Patron de conception, solution réutilisable à un problème récurrent en conception logicielle.

Jackson Bibliothèque Java pour la sérialisation/désérialisation JSON.

JVM (Java Virtual Machine) Machine virtuelle Java permettant l'exécution du bytecode Java sur différentes plateformes.

ORM (Object-Relational Mapping) Technique de programmation permettant de mapper des objets avec une base de données relationnelle.

SOLID Acronyme pour cinq principes de conception orientée objet : Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion.

Thread-Safety Propriété d'un code pouvant être exécuté simultanément par plusieurs threads sans causer d'incohérences.

UML (Unified Modeling Language) Langage de modélisation unifié pour la conception de systèmes orientés objet.

6.4 Bibliographie et Références

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.
2. Bloch, J. (2017). *Effective Java* (3rd ed.). Addison-Wesley Professional.
3. Martin, R. C. (2017). *Clean Architecture : A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
4. Oracle Corporation. (2024). *Java Platform, Standard Edition Documentation*. Retrieved from https://docs.oracle.com/javase/
5. FasterXML. (2024). *Jackson Project Documentation*. Retrieved from https://github.com/FasterXML/jackson
6. Fowler, M. (2003). *UML Distilled : A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley Professional.

Ce rapport constitue une base solide pour comprendre, maintenir et faire évoluer le système de gestion d'événements. Pour toute question ou approfondissement, veuillez me contacter.