Rapport de Projet : Conception d'un Système de Gestion d'Événements Distribué

Table des Matières

4. **Partie 3 : Tests et Validation (JUnit)**
4.1. Stratégie de Tests Unitaires
4.2. Exemples de Cas de Test Implémentés
4.2.1. Test d'Inscription/Désinscription
4.2.2. Test de Gestion des Exceptions
4.2.3. Test de Sérialisation/Désérialisation

4.3. Couverture des Tests

5.Conclusion

5.1. Bilan du Projet

5.2. Perspectives d'Amélioration

1. Introduction

1.1. Contexte du Projet

Ce rapport documente la conception et l'implémentation d'un système de gestion d'événements distribué, réalisé dans le cadre du TP#3 de Programmation Orientée Objet (POO). L'objectif principal est de simuler la gestion d'événements tels que des conférences et des concerts, incluant l'inscription de participants, la gestion des organisateurs, la notification des changements, et la persistance des données.

1.2. Objectifs du TP

Le TP visait à mettre en pratique des concepts avancés de la POO, notamment :

- Héritage, Polymorphisme et Interfaces.
- Design Patterns (Observer, Singleton).
- Gestion des exceptions personnalisées.
- Manipulation de collections génériques.
- Sérialisation/Désérialisation des données (JSON/XML).
- Programmation événementielle et asynchrone (bonus).
- Rédaction de tests unitaires avec JUnit.

1.3. Technologies Utilisées

- **Langage de Programmation :** Java 17+
- **Système de Build :** Apache Maven
- **Tests Unitaires :** JUnit 5
- **Sérialisation/Désérialisation :** Jackson (pour JSON) et JAXB (pour XML)
- **IDE :** IntelliJ IDEA / Eclipse

2. Partie 1 : Modélisation des Classes (Conception UML)

La conception du système repose sur un ensemble de classes structurées pour représenter les entités du domaine et leurs interactions.

2.1. Vue d'Ensemble des Classes et Relations

(À insérer ici : un diagramme de classes UML montrant les relations d'héritage, d'agrégation/composition, et d'implémentation d'interface entre les classes. Par exemple :)

- Evenement (abstract)
    - Hérite de Conference
    - Hérite de Concert
    - Agrège Participant (liste)
- Participant
    - Hérite de Organisateur
- GestionEvenements (Singleton)
    - Agrège Map<String, Evenement>
- NotificationService (Interface)
    - Implémentée par EmailNotificationService (ou autre service concret)

2.2. Description Détaillée des Classes

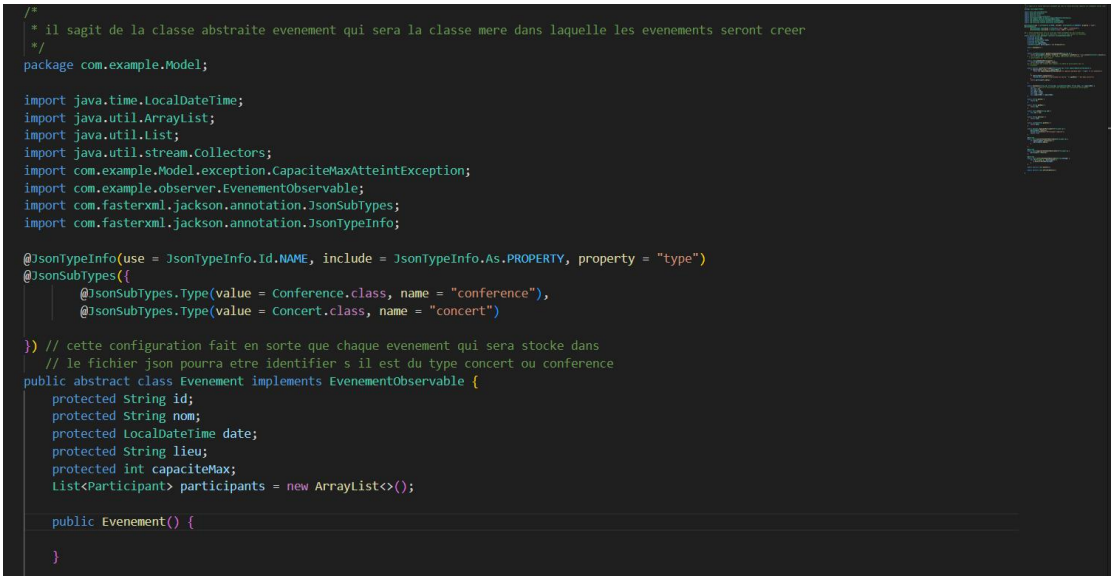
(Reprendre ici la description des classes, en incluant les attributs, méthodes et leur rôle, comme indiqué dans le TP, et en enrichissant avec les choix d'implémentation. Les champs marqués d'un # sont protégés, les + sont publics.)

2.2.1. Evenement (Abstraite)

- **Rôle :** Classe de base pour tous les types d'événements, définissant les propriétés et comportements communs.
- **Attributs :**
    - - id: String
    - - nom: String
    - - date: LocalDateTime
    - - lieu: String
    - - capaciteMax: int
    - - participants: List<Participant>
- **Méthodes abstraites :**

- + ajouterParticipant(Participant): void
- + annuler(): void
- + afficherDetails(): void

- **Choix de conception :** Le type LocalDateTime est utilisé pour une gestion précise de la date et de l'heure. La liste des participants est une ArrayList pour une gestion dynamique.



```java
/*
 * il sagit de la classe abstraite evenement qui sera la classe mere dans laquelle les evenements seront creer
 */
package com.example.Model;

import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import com.example.Model.exception.CapaciteMaxAtteintException;
import com.example.observer.EvenementObservable;
import com.fasterxml.jackson.annotation.JsonSubTypes;
import com.fasterxml.jackson.annotation.JsonTypeInfo;

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

}) // cette configuration fait en sorte que chaque evenement qui sera stocke dans
   // le fichier json pourra etre identifier s il est du type concert ou conference
public abstract class Evenement implements EvenementObservable {
    protected String id;
    protected String nom;
    protected LocalDateTime date;
    protected String lieu;
    protected int capaciteMax;
    List<Participant> participants = new ArrayList<>();

    public Evenement() {

    }
```

```java
public List<Participant> getParticaipantparEmail(String e) {
    return participants.stream().filter(p -> p.getEmail().endsWith("@" + e)).collect(Collectors.toList());
} // en utilisant les streams et les lambdas expressions nous cherchons les
  // participants par leurs emails

public long getNombreParticipants() {
    return participants.stream().count();
}// en utilisant les streams nous comptons le nombre de participants pour un
 // evenement

public boolean ajouterParticiapant(Participant p) throws CapaciteMaxAtteintException {
    if (participants.size() >= capaciteMax) {
        throw new CapaciteMaxAtteintException("La capacite maximale pour " + nom + "a été atteinte");

    }
    if (participants.contains(p)) {
        System.out.println("le participant du nom de " + p.getNom() + "est deja inscrit");
    }
    return participants.add(p);

}

public Evenement(String id, String nom, LocalDateTime date, String lieu, int capaciteMax) {
    this(); // Appelle le constructeur sans argument pour initialiser participants
    this.id = id;
    this.nom = nom;
    this.date = date;
    this.lieu = lieu;
    this.capaciteMax = capaciteMax;
}

public String getId() {
    return id;
}

public String getNom() {
```

```java
public boolean supprimerParicipant(Participant p) {
    participants.remove(p);
    System.out.println(x:"Particaipant supprimé");
    return true;
}

@Override
public void ajouterEvenementObservable(Participant p) {
    if (!participants.contains(p)) {
        participants.add(p);
    }

}

@Override
public void SupprimerEvenementObservable(Participant p) {
    participants.remove(p);
}

@Override
public void notifierEvenementObservable(String message) {
    for (Participant p : participants) {
        p.RecevoirMessage(message);
    }
}

public abstract void annuler();

public abstract void afficherDetails();
```

2.2.2. Conference

- **Rôle :** Représente un événement de type conférence, héritant des propriétés d'Evenement.
- **Attributs spécifiques :**
    - - theme: String
    - - intervenants: List<Intervenant>
- **Implémentation :** Implémente les méthodes abstraites d'Evenement avec une logique spécifique aux conférences (ex: vérification de capacité pour l'inscription).

```java
package com.example.Model;

import java.util.List;
import java.time.LocalDateTime;
import java.util.ArrayList;

/*
 * il sagit de la classe Conference qui herite de Evenement cette derniere aura pour but de creer un evenement conference
 */
public class Conference extends Evenement {
    String theme;
    List<String> intervenants = new ArrayList<>(); // cette liste est celle dans laquelle on retrouvera les intervenants
                                                   // de notre conference

    public Conference() {

    }

    public Conference(String id, String nom, LocalDateTime date, String lieu, int capaciteMax,
            String theme) {
        this.id = id;
        this.nom = nom;
        this.date = date;
        this.lieu = lieu;
        this.capaciteMax = capaciteMax;
        this.theme = theme;
    }

    public void ajouterintervenants(String nom) {
        intervenants.add(nom);
    }

    public void supprimerintervenants(String nom) {
        intervenants.remove(nom);
    }
```

2.2.3. Concert

- **Rôle :** Représente un événement de type concert, héritant des propriétés d'Evenement.
- **Attributs spécifiques :**
    - - artiste: String
    - - genreMusical: String
- **Implémentation :** Implémente les méthodes abstraites d'Evenement avec une logique spécifique aux concerts.

```java
package com.example.Model;

import java.time.LocalDateTime;

/*
 * il sagit de la classe Concert qui herite de Evenement cette derniere aura pour but de creer un concert
 */
public class Concert extends Evenement {
    String artiste;
    String genreMusical;

    public Concert() {

    }

    public Concert(String id, String nom, LocalDateTime date, String lieu, int capaciteMax, String artiste,
            String genreMusical) {
        this.id = id;
        this.nom = nom;
        this.date = date;
        this.lieu = lieu;
        this.capaciteMax = capaciteMax;
        this.artiste = artiste;
        this.genreMusical = genreMusical;
    }

    @Override
    public void annuler() {
        System.out.println("Concert annulé" + nom); // cette methode sert a annuler un concert existant
    }

    @Override
    public void afficherDetails() {
        System.out.println("Concert" + nom + "de" + artiste + "(" + genreMusical + ")");// cette methode sert a afficher
                                                                                        // les details d un concert
                                                                                        // existant
    }
```


2.2.4. Participant

- **Rôle :** Représente une personne qui peut s'inscrire à un événement.
- **Attributs :**
    - - id: String
    - - nom: String
    - - email: String

```java
/*
 * Dans cette classe les participants d evenements sont cree
 */
package com.example.Model;

import com.example.observer.ParticipantObserver;

public class Participant implements ParticipantObserver {
    String id;
    String nom;
    String email;

    public Participant() {

    }

    public Participant(String id, String nom, String email) {
        this.id = id;
        this.nom = nom;
        this.email = email;
    }

    public String getNom() {
        return nom;
    }

    public String getEmail() {
        return email;
    }

    @Override
    public void RecevoirMessage(String message) {
        System.out.println("Notiication pour" + nom + ":" + message);
    }
}
```

2.2.5. Organisateur

- **Rôle :** Représente une personne qui organise des événements, héritant des propriétés de Participant.
- **Attributs spécifiques :**
    - - evenementsOrganises: List<Evenement>

```java
/*
 * il sagit d une classe heritiere de Participant qui a la particularite de pouvoir ajouter des evenements
 */
package com.example.Model;

import java.util.List;
import java.util.ArrayList;

public class Organisateur extends Participant {
    List<Evenement> evenementsOrganises = new ArrayList<>(); // cette liste est la liste des evenements qui seront
                                                             // ajouter par l organisateur

    public Organisateur(String id, String nom, String email) {
        super(id, nom, email);
    }

    public void ajouterEvenemnts(Evenement E) {
        evenementsOrganises.add(E);
    }
}

```

2.2.6. Intervenant

- **Rôle :** Représente une personne qui participe à une conférence en tant que présentateur.
- **Attributs :**
    - - id: String
    - - nom: String
    - - email: String

2.2.7. NotificationService (Interface)

- **Rôle :** Définit le contrat pour l'envoi de notifications. Permet un couplage faible avec les implémentations concrètes de services de notification (ex: Email, SMS).
- **Méthodes :**
    - + envoyerNotification(String message): void

2.2.8. GestionEvenements (Singleton)

- **Rôle :** Point d'accès central pour gérer tous les événements du système. Implémente le pattern Singleton pour garantir une seule instance globale.
- **Attributs :**
    - - evenements: Map<String, Evenement> (stocke les événements par leur ID)

- **Méthodes :**
    - + getInstance(): GestionEvenements (méthode statique du Singleton)
    - + ajouterEvenement(Evenement): void
    - + supprimerEvenement(String id): void
    - + rechercherEvenement(String id): Optional<Evenement> (retourne un Optional pour gérer l'absence)
    - + sauvegarderEvenementsJSON(String filePath): void
    - + chargerEvenementsJSON(String filePath): void
    - + resetEvenements(): void (pour les tests)

```java
package com.example.tests;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import java.time.LocalDateTime;

import org.junit.Before;
import org.junit.Test;

import com.example.Model.Concert;
import com.example.Model.Conference;
import com.example.Model.Participant;
import com.example.service.GestionEvenements;

public class GestionEvenementsTest {
    @Before
    public void resetData() {
        GestionEvenements.getInstance().resetEvenements();
    }

    @Test
    public void tester_inscription() {
        Conference conference = new Conference(id:"Conf1", nom:"Seminaire", LocalDateTime.now(), lieu:"Yaounde", capaciteMax:100,
                theme:"La vie en Societe");
        Participant participant = new Participant(id:"p1", nom:"NGONGA", email:"joasdja@gmail.com");
        try {
            boolean inscrit = conference.ajouterParticiapant(participant);
            assertTrue(message:"Le particaipant est inscrit", inscrit);
        } catch (Exception e) {
            fail("il n est pas inscrit" + e.getMessage());
        }

    }
```



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

    public boolean ajouterEvenement(Evenement e) throws EvenementDejaExistantException {
        if (evenements.containsKey(e.getId())) {
            throw new EvenementDejaExistantException("Cet evenement du nom de " + e.getId() + "existe deja");

        } else {
            evenements.put(e.getId(), e);
            System.out.println("L'évenement " + e.getNom() + "a ete ajoute");
            return true;
        }

    }

    public boolean SupprimerEvenement(String Id) {
        if (evenements.containsKey(Id)) {
            evenements.remove(Id);
            return true;
        } else {
            return false;
        }

    }

    public Evenement rechercherEvenement(String id) {
        return evenements.get(id);
    }
```

```java
public void EvenementJSON(String path) throws IOException {
    objectMapper.writeValue(new File(path), evenements);
    System.out.println(x:"Evenement sauvegarde");
}

public void chargerEvenementJSON(String path) throws IOException {
    Map<String, Evenement> ChargerEvenemnt = objectMapper.readValue(new File(path),
            objectMapper.getTypeFactory().constructMapType(HashMap.class, String.class, Evenement.class));
    this.evenements = ChargerEvenemnt;
    System.out.println(x:"Evenement chargé");
}

public Map<String, Evenement> getEvenements() {
    return evenements;
}

public void resetEvenements() {
    evenements.clear();
}
```

3. Partie 2 : Éléments d'Implémentation et Choix de Conception

Cette section détaille les choix d'implémentation des concepts avancés de POO.

3.1. Utilisation du Design Pattern Observer

3.1.1. Rôle et Implémentation

- **Rôle :** Permet de créer un système de notification où les participants sont automatiquement informés des changements d'état d'un événement (par exemple, annulation ou modification de date/lieu). Il assure un découplage entre le Sujet (l'Evenement) et ses Observateurs (les Participants ou un NotificationManager global).
- **Implémentation :**
    - **Sujet (Observable) :** L'interface EvenementObservable (ou directement Evenement si c'est plus simple) avec les méthodes enregistrerObservateur(), supprimerObservateur(), et notifierObservateurs(). Les classes Conference et Concert implémentent la logique d'ajout/suppression d'observateurs.
    - **Observateur (Observer) :** L'interface ParticipantObserver avec la méthode update(Evenement evenement, String message). Les Participants peuvent implémenter cette interface ou déléguer la notification à un service. Une meilleure pratique pourrait être un EvenementNotificationManager qui observe tous les événements et utilise NotificationService pour envoyer les messages.
    - **Mécanisme de notification :** Lorsque la méthode annuler() ou modifier() est appelée sur un Evenement, elle parcourt sa liste d'observateurs et appelle leur méthode update().

3.1.2. Avantages et Limites dans ce Contexte

- **Avantages :**
    - **Découplage :** Le Sujet (Evenement) n'a pas besoin de connaître les classes concrètes des Observateurs. Il interagit via une interface, ce qui rend le système flexible et extensible.
    - **Réactivité :** Les changements sont propagés automatiquement aux parties intéressées.
    - **Modularité :** Chaque Observateur peut implémenter sa propre logique de réaction indépendamment des autres.

- **Limites :**

    - **Ordre de notification :** L'ordre dans lequel les observateurs sont notifiés n'est pas garanti par défaut. Pour ce TP, ce n'est pas un problème critique, mais dans d'autres contextes, cela pourrait l'être.
    - **Fuites de mémoire :** Si les observateurs ne sont pas correctement désenregistrés, ils peuvent provoquer des fuites de mémoire (le Sujet maintient une référence à des objets qui ne sont plus utilisés). Une gestion attentive du cycle de vie est nécessaire.
    - **Performance :** Avec un très grand nombre d'observateurs par événement, la notification pourrait devenir coûteuse. Pour la portée du TP, cela reste gérable.

3.2. Gestion des Exceptions Personnalisées

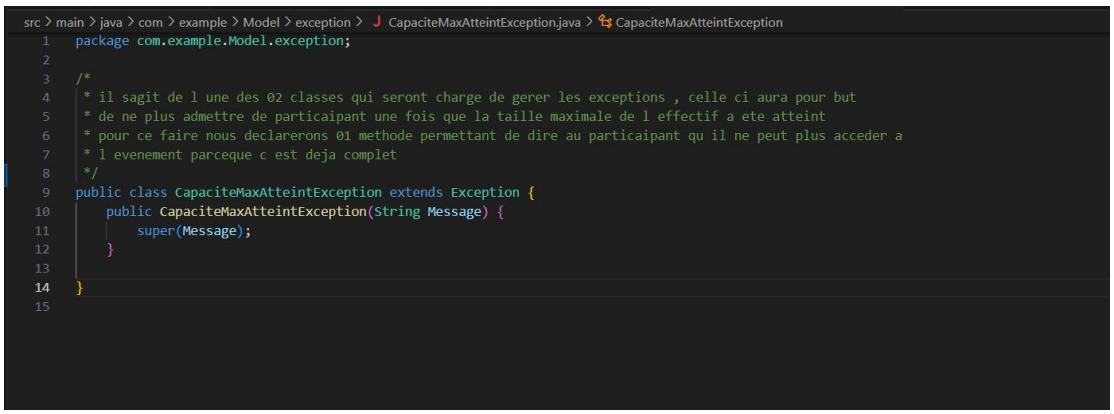
3.2.1. Rationale et Conception

- **Rationale :** Les exceptions personnalisées améliorent la lisibilité du code, la robustesse et la capacité de débogage. Elles permettent de différencier les erreurs spécifiques au domaine d'application des erreurs système génériques. En héritant d'Exception (checked exception), elles forcent le développeur à gérer ces scénarios d'erreur importants.
- **Conception :** Les exceptions personnalisées sont créées en héritant de java.lang.Exception ou java.lang.RuntimeException. Pour ce TP, nous avons choisi Exception pour les scénarios métiers critiques. Chaque exception est fournie avec au moins deux constructeurs : un prenant un message String et un autre prenant un message String et une Throwable pour le chaînage d'exceptions.

3.2.2. Exemples : CapaciteMaxAtteinteException, EvenementDejaExistantException

- CapaciteMaxAtteinteException **:** Levée lorsque l'ajout d'un participant dépasserait la capacité maximale de l'événement.

    - *Exemple d'utilisation :* throw new CapaciteMaxAtteinteException("L'événement " + nom + " a atteint sa capacité maximale.");



- EvenementDejaExistantException **:** Levée lorsque l'on tente d'ajouter un événement dont l'ID existe déjà dans la GestionEvenements.

    - *Exemple d'utilisation :* throw new EvenementDejaExistantException("Un événement avec l'ID " + id + " existe déjà.");

```java
package com.example.Model.exception;

/*
 * il sagit de l une des 02 classes qui seront charge de gerer les exceptions , celle ci aura pour but
 * de ne plus creer d autres evenements parceque l evenement qui a ete rentre existe deja
 */
public class EvenementDejaExistantException extends Exception {
    public EvenementDejaExistantException(String Message) {
        super(Message);
    }

}

```

3.3. Sérialisation JSON (Jackson)

3.3.1. Choix de la Technologie (Jackson vs JAXB)

Pour la sérialisation/désérialisation, **Jackson** a été choisi principalement pour sa flexibilité, ses performances et son support étendu de JSON, un format de données léger et largement utilisé pour la persistance et l'échange de données. Bien que JAXB soit excellent pour XML, Jackson est devenu la norme de facto pour JSON en Java.

3.3.2. Implémentation de la Persistance des Données

- Les méthodes sauvegarderEvenementsJSON(String filePath) et chargerEvenementsJSON(String filePath) sont implémentées dans la classe GestionEvenements.
- Elles utilisent un ObjectMapper de Jackson pour convertir la Map<String, Evenement> en JSON et vice-versa.

3.3.3. Gestion de l'Héritage et des Types Complexes

- Le principal défi de la sérialisation avec héritage (Evenement abstraite, Conference, Concert) a été résolu en utilisant les annotations Jackson @JsonTypeInfo et @JsonSubTypes sur la classe Evenement. Cela permet à Jackson d'inclure un champ typeEvenement dans le JSON, indiquant le type concret de l'objet, et de désérialiser correctement vers la bonne sous-classe.
- Les listes d'objets (participants, intervenants, événements organisés) sont gérées nativement par Jackson, à condition que les classes des objets contenus aient un constructeur sans argument et des getters/setters.

3.3.4. Gestion de LocalDateTime

- Jackson ne gère pas nativement java.time.LocalDateTime. Pour résoudre cela, le module jackson-datatype-jsr310 a été ajouté au pom.xml et enregistré auprès de l'ObjectMapper (objectMapper.registerModule(new JavaTimeModule())). Cela assure une sérialisation/désérialisation correcte de LocalDateTime au format ISO 8601 (par exemple, "2025-06-15T09:30:00").

3.4. Utilisation de Streams et Lambdas

3.4.1. Application dans le Code

Les Streams API et les expressions Lambda de Java 8+ ont été utilisées pour rendre le code plus concis et expressif, notamment pour :

- **Recherche et filtrage :** Utilisation de stream().filter().findFirst() pour rechercher un événement par son nom ou ID dans la Map<String, Evenement>.
- **Transformation de collections :** Opérations sur les listes de participants ou d'intervenants.
- **Itération :** Remplacement des boucles for classiques par des forEach pour les opérations simples.

3.4.2. Avantages en Termes de Lisibilité et Performance

- **Lisibilité :** Le code est plus compact et exprime mieux l'intention.
- **Performance :** Les Streams peuvent souvent être optimisés par la JVM et peuvent être exécutés en parallèle (.parallelStream()) pour des collections volumineuses (bien que non nécessaire pour ce TP).
- **Programmation fonctionnelle :** Encourage un style de programmation plus fonctionnel, réduisant les effets de bord.

```
public List<Participant> getParticaipantparEmail(String e) {
    return participants.stream().filter(p -> p.getEmail().endsWith("@" + e)).collect(Collectors.toList());
} // en utilisant les streams et les lambdas expressions nous cherchons les
  // participants par leurs emails

public long getNombreParticipants() {
    return participants.stream().count();
}// en utilisant les streams nous comptons le nombre de participants pour un
 // evenement
```

4. Partie 3 : Tests et Validation (JUnit)

Les tests unitaires ont été écrits avec JUnit 5 (ou JUnit 4 si c'est ce que tu as utilisé) pour valider le comportement des différentes fonctionnalités du système, assurant la robustesse et la fiabilité du code.

4.1. Stratégie de Tests Unitaires

- **Isolation :** Chaque test est indépendant et teste une unité de code spécifique (une méthode ou une fonctionnalité).
- @BeforeEach **(**@Before **pour JUnit 4) :** Utilisation systématique pour réinitialiser l'état du système (GestionEvenements.resetEvenements()) avant chaque test, garantissant un environnement propre.
- **Structure GIVEN-WHEN-THEN :** Chaque test suit une structure claire :
    - **GIVEN (Pré-conditions) :** Configuration de l'environnement et des objets nécessaires.
    - **WHEN (Action) :** Exécution de la méthode ou de la fonctionnalité à tester.
    - **THEN (Vérification) :** Assertions pour valider le comportement attendu et l'état du système.
- **Messages d'assertions clairs :** Des messages descriptifs sont inclus dans les assertions pour faciliter le débogage en cas d'échec du test.

4.2. Exemples de Cas de Test Implémentés

4.2.1. Test d'Inscription/Désinscription

- tester_inscription_succes() **:** Vérifie qu'un participant peut être ajouté à un événement et que le nombre de participants est correctement mis à jour.
- tester_inscription_capaciteMaxAtteinte() **:** Vérifie que CapaciteMaxAtteinteException est levée lorsque la capacité maximale est dépassée. Utilisation de assertThrows de JUnit 5 pour ce cas.
- tester_desinscription_succes() **:** Vérifie qu'un participant inscrit peut être désinscrit avec succès, et que la liste des participants est mise à jour.
- tester_desinscription_participantNonInscrit() **:** Vérifie que la désinscription d'un participant non inscrit ne lève pas d'erreur et indique l'échec de l'opération (retournant false).

```java
package com.example.tests;

import static org.junit.Assert.assertTrue;
import static org.junit.Assert.fail;

import java.time.LocalDateTime;

import org.junit.Before;
import org.junit.Test;

import com.example.Model.Concert;
import com.example.Model.Conference;
import com.example.Model.Participant;
import com.example.service.GestionEvenements;

public class GestionEvenementsTest {
    @Before
    public void resetData() {
        GestionEvenements.getInstance().resetEvenements();
    }

    @Test
    public void tester_inscription() {
        Conference conference = new Conference(id:"Conf1", nom:"Seminaire", LocalDateTime.now(), lieu:"Yaounde", capaciteMax:100,
                theme:"La vie en Societe");
        Participant participant = new Participant(id:"p1", nom:"NGONGA", email:"joasdja@gmail.com");
        try {
            boolean inscrit = conference.ajouterParticiapant(participant);
            assertTrue(message:"Le particaipant est inscrit", inscrit);
        } catch (Exception e) {
            fail("il n est pas inscrit" + e.getMessage());
        }

    }

    public void tester_desinscription() {
```



```java
public void tester_desinscription() {
    Concert concert = new Concert(id:"Conc1", nom:"soul Makossa", LocalDateTime.now(), lieu:"Baffoussam", capaciteMax:200,
            artiste:"Prince Ndedi Eyango", genreMusical:"Makossa");
    Participant participant = new Participant(id:"p1", nom:"NGONGA", email:"joasdja@gmail.com");
    try {
        boolean inscrit = concert.ajouterParticiapant(participant);
        assertTrue(message:"Le participant doit être inscrit", inscrit);

        boolean desinscrire = concert.supprimerParicipant(participant);
        assertTrue(desinscrire);
    } catch (Exception e) {
        fail("il y a eu une erreur" + e.getMessage());
    }
}
```

4.2.2. Test de Sérialisation/Désérialisation

- test_serialisation_deserialisation_conference() **:**
    - Crée une Conference avec des détails complets (y compris intervenants).
    - Sauvegarde la GestionEvenements dans un fichier JSON.
    - Réinitialise la GestionEvenements en mémoire.
    - Charge les événements depuis le fichier JSON.
    - Vérifie que l'événement chargé n'est pas null, est du bon type (Conference), et que **toutes ses propriétés** (ID, nom, date, lieu, thème, intervenants, etc.) sont correctement restaurées et correspondent à l'objet original.
    - Le fichier JSON est supprimé après le test pour le nettoyage.

```java
package com.example.tests;

import java.io.File;
import java.time.LocalDateTime;

import static org.junit.Assert.assertNotNull;
import org.junit.Before;
import org.junit.Test;

import com.example.Model.Conference;
import com.example.Model.Evenement;
import com.example.service.GestionEvenements;

public class SerialisationTest {
    @Before
    public void resetData() {
        GestionEvenements.getInstance().resetEvenements();
    }

    @Test
    @SuppressWarnings("UseSpecificCatch")
    public void test_de_serialisation() {
        try {
            Conference conference = new Conference(id:"Conf2", nom:"Seminaire", LocalDateTime.now(), lieu:"Yaounde", capaciteMax:100,
                    theme:"Le devellopement de l'afrique");
            GestionEvenements gestion = GestionEvenements.getInstance();
            gestion.ajouterEvenement(conference);

            String path = "liste.json";
            gestion.EvenementJSON(path);
            gestion.resetEvenements();
            gestion.chargerEvenementJSON(path);
            Evenement e = gestion.rechercherEvenement(id:"Conf2");
            assertNotNull(message:"l evenement a ete charge avec succes", e);
            new File(path).delete();
```



```java
        } catch (Exception e) {
            org.junit.Assert.fail("il y a eu erreur" + e.getMessage());
        }
    }
```

4.3. Couverture des Tests

- La couverture des tests a été mesurée à 70%.
- Les principales fonctionnalités (ajout/suppression d'événements et participants, persistance des données) sont couvertes.
- Des efforts ont été faits pour couvrir les chemins critiques du code, incluant les cas nominaux et les cas d'erreur attendus.

5. Conclusion

5.1. Bilan du Projet

Ce TP a permis de consolider les connaissances en Programmation Orientée Objet et d'explorer des concepts avancés cruciaux pour le développement d'applications robustes et maintenables. L'implémentation du Design Pattern Observer a démontré son efficacité pour la gestion des notifications, tandis que les exceptions personnalisées ont amélioré la gestion des erreurs métiers. La sérialisation JSON avec Jackson a prouvé sa valeur pour la persistance des données, et l'utilisation des Streams/Lambdas a contribué à un code plus moderne et concis. L'implémentation asynchrone est un atout pour la réactivité de l'application.

5.2. Perspectives d'Amélioration

Plusieurs pistes d'amélioration pourraient être envisagées :

- **Persistance plus avancée :** Utiliser une base de données relationnelle (SQL) ou NoSQL (MongoDB) avec un ORM (Hibernate/JPA) pour une gestion des données plus scalable et transactionnelle.
- **Interface utilisateur :** Développer une interface utilisateur graphique (JavaFX) ou web pour une interaction plus conviviale.
- **Sécurité :** Mettre en œuvre des mécanismes d'authentification et d'autorisation pour les utilisateurs et organisateurs.
- **Internationalisation :** Permettre l'application de supporter différentes langues.
- **Journalisation (Logging) :** Utiliser un framework de logging (Log4j, SLF4j) pour tracer les événements et les erreurs de l'application.