REPULIQUE DU CAMEROUN
****
Paix – Travail – Patrie
******
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR
****
UNIVERSITE DE YAOUNDE I
****
ECOLE NATIONAL SUPERIEURE POLYTECHNIQUE DE YAOUNDE
****
DEPARTEMENT DU GENIE INFORMATIQUE

REPUBLIC OF CAMEROON
****
Peace – Work – Fatherland
****
MINISTRY OF HIGHER EDUCATION
****
UNIVERSITY OF YAOUNDE I
****
NATIONAL ADVANCED SCHOOL OF ENGINEERING
****
DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

RAPPORT DU TP3 FINAL P.O.O : APPLICATION DE GESTION DES EVENEMENT

Proposer par: TCHAPDA PIEME ALAN CHANEL
Matricule : 24P761
Classe : 3GI
Sous la direction de: Dr KUNGNE
Année académique: 2024 – 2025

Table des matières

1- INTRODUCTION

Dans le cadre de ce projet, il s'agissait de concevoir et développer un système de gestion d'événements permettant d'administrer des conférences, des concerts et autres manifestations similaires. Ce projet met en œuvre les principes fondamentaux et avancés de la **Programmation Orientée Objet (POO)**, notamment l'héritage, le polymorphisme, et l'utilisation de design patterns tels que **Singleton**, **Observer**, **Factory**, et **Strategy**.

L'objectif principal était de réaliser un système capable d'enregistrer et de gérer des participants, des organisateurs, ainsi que de gérer les notifications relatives aux événements, de manière synchrone et asynchrone. La persistance des données a été assurée par la **sérialisation et désérialisation en format JSON**, en utilisant la bibliothèque **Jackson**.

Le projet a été développé en **Java 11**, avec l'utilisation de **Maven** pour la gestion des dépendances, et de **JUnit** pour la création et l'exécution des tests unitaires. Ces choix technologiques assurent une **structure modulaire et robuste**, garantissant une maintenance et une évolutivité facilitées.

2-CONCEPTION

DIAGRAMME UML

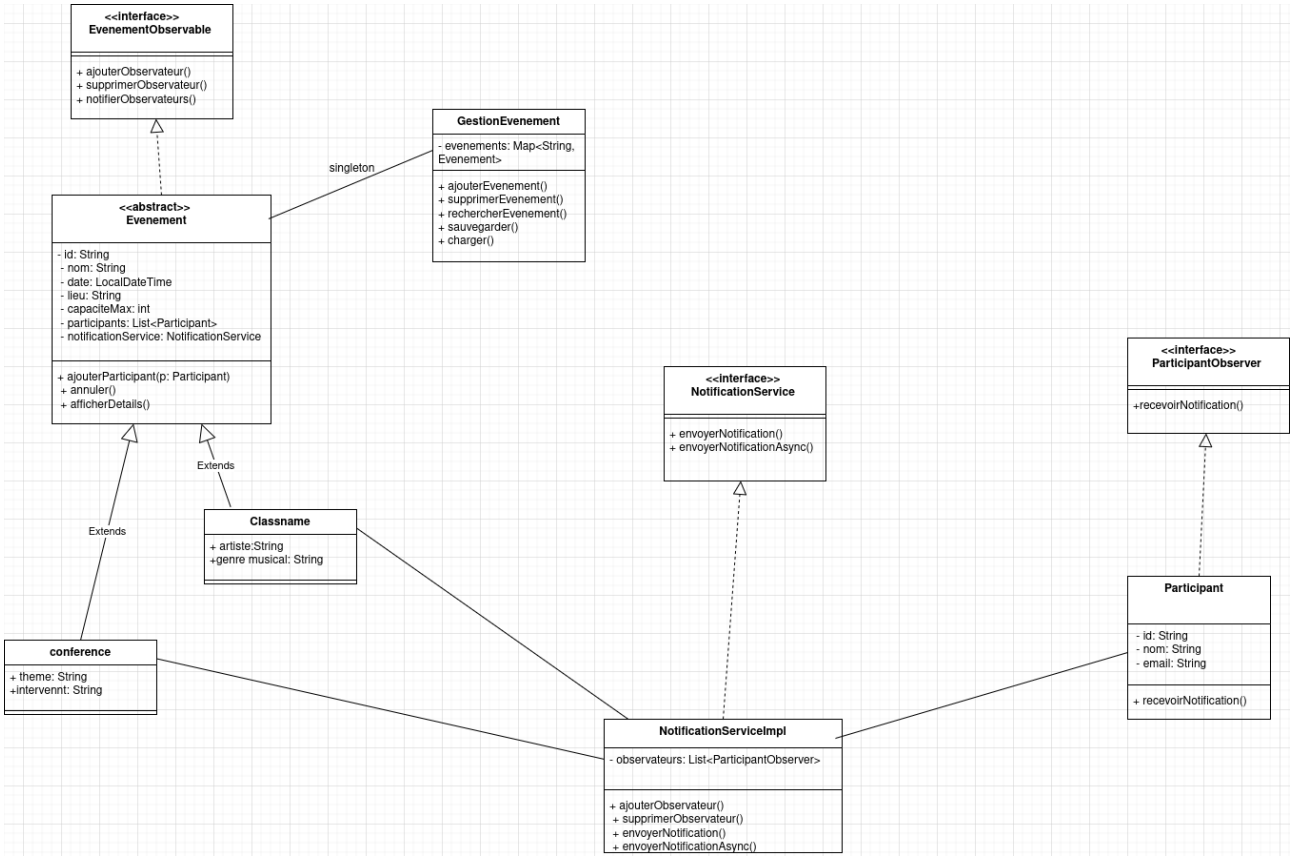
Le système de gestion d'événements a été conçu selon les principes de la **programmation orientée objet**, afin de garantir une architecture claire, modulaire et extensible. La conception repose sur une modélisation UML mettant en évidence les principales entités du système et leurs relations.

- Evenement (abstraite), avec ses sous-classes Conference et Concert

-Participant et Organisateur

-GestionEvenements (singleton),

-Les interfaces NotificationService, EvenementObservable, ParticipantObserver.



Parmi les classes principales, on retrouve la classe abstraite Evenement, qui représente un événement générique et sert de base pour les classes Conference et Concert. Ces dernières héritent des propriétés communes d'un événement et ajoutent des attributs spécifiques tels que le thème ou le genre musical. La classe Participant permet de gérer les personnes inscrites à un événement, tandis que Organisateur, qui hérite de Participant, gère l'organisation et l'ajout d'événements.

Singleton:

Un point clé de la conception réside dans l'utilisation du **design pattern Singleton**, implémenté par la classe GestionEvenements. Cette classe assure la centralisation et le contrôle de tous les événements du système, garantissant qu'une seule instance soit accessible.

```
/// ////// singleton
public static synchronized GestionEvenements getInstance(){  2 usages  GAUSS-TPAC
    if (INSTANCE == null){
        INSTANCE= new GestionEvenements();
    }
    return INSTANCE;
}
///////
```

Explication:

- **Singleton** : garantit une seule instance globale de la classe (ici GestionEvenements).
- **Synchronisation** : rend la méthode getInstance() thread-safe pour éviter des créations multiples.
- **Pourquoi ?** : Cela centralise la gestion des événements dans le système, et évite les incohérences.

Observer:

De plus, le **design pattern Observer** a été mis en place pour permettre aux participants d'être automatiquement notifiés en cas de modification ou d'annulation d'un événement. Ce mécanisme favorise la réactivité et l'interactivité du système, et assure une communication fluide entre les événements et les participants.

```
1   package service;
2
3   public interface EvenementObservable {  2 usages  3 implementations  GAUSS-TPAC
4       void notifierObservateurs(String message);  no usages  1 implementation  GAUSS-TPAC
5   }
```

Pour les évenements

```
1   package service;
2
3   public interface ParticipantObserver{  6 usages  2 implementations  GAUSS-TPAC
4
5       void recevoirNotification(String message);  2 usages  1 implementation  GAUSS-TPAC
6   }
```

pour les Participants

Explication:

- Ces interfaces définissent le **design pattern Observer** :
- EvenementObservable est l'émetteur des notifications (Evenement).
- ParticipantObserver est le récepteur (Participant).
- **Pourquoi ?** : Cela permet d'avoir un système flexible, où les participants peuvent être notifiés automatiquement.

Persistance de données

Pour la persistance des données, la sérialisation en format JSON a été adoptée grâce à la bibliothèque **Jackson**, permettant de sauvegarder et de charger la liste des événements. Ce choix garantit la portabilité des données et la compatibilité avec d'autres systèmes. Enfin, une couche asynchrone a été ajoutée via l'utilisation de CompletableFuture afin de simuler l'envoi différé des notifications, renforçant l'aspect distribué du système.

```
@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")
})

public abstract class  Evenement implements EvenementObservable {
    private  String id;  3 usages
```

Explication:

-**Annotation** @JsonTypeInfo **et** @JsonSubTypes : elles permettent à **Jackson** de gérer la sérialisation/désérialisation des objets polymorphiques (ici Evenement est abstraite, ses sous-classes sont Conference, Concert

-Elles indiquent à Jackson de **sauvegarder le type de sous-classe dans le JSON**, pour savoir quelle classe instancier lors de la lecture.

-**Pourquoi ?** : Sans ces annotations, Jackson ne pourrait pas désérialiser correctement les objets hérités.

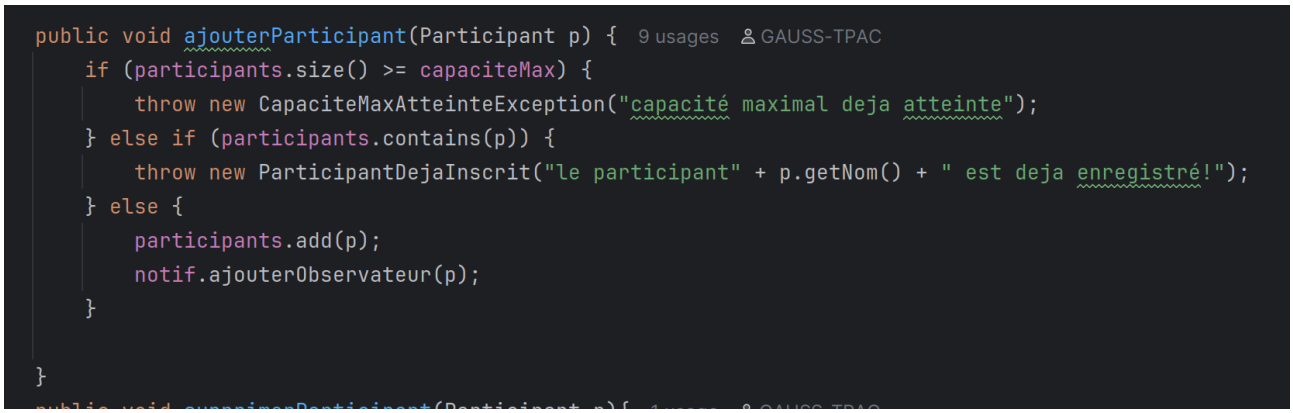
3- IMPLENTATION

Le projet a été développé en **Java 11**, avec l'utilisation de **Maven** pour la gestion des dépendances, garantissant une structure modulaire et facile à maintenir. L'implémentation s'appuie sur l'architecture définie par la modélisation UML, traduite en classes concrètes et interfaces.

La classe abstraite Evenement a été conçue pour encapsuler les propriétés et comportements communs à tous les événements, tels que l'ajout de participants, l'affichage des détails, ou encore

l’annulation d’un événement. Les classes Conference et Concert, dérivées d’Evenement, ajoutent des attributs spécifiques tout en héritant des méthodes génériques.

Ajout d’un participant avec vérification de capacité



```
public void ajouterParticipant(Participant p) {  9 usages   GAUSS-TPAC
    if (participants.size() >= capaciteMax) {
        throw new CapaciteMaxAtteinteException("capacité maximal deja atteinte");
    } else if (participants.contains(p)) {
        throw new ParticipantDejaInscrit("le participant" + p.getNom() + " est deja enregistré!");
    } else {
        participants.add(p);
        notif.ajouterObservateur(p);
    }

}
```

Explication :

- **Vérification métier** : contrôle de la capacité maximale et doublons avant l’inscription.
- **Exception personnalisée** : levée d’exception spécifique pour signaler le problème.
- **Pourquoi ?** : renforce la robustesse et la clarté des erreurs dans le système.

Méthodes de sérialisation/désérialisation JSON

serialization

```java
public void  sauvegarder(Map<String, Evenement> evenementMap, String chemin) throws IOException {  1 us
    ObjectMapper mapperW= new ObjectMapper();

    mapperW.registerModule(new JavaTimeModule());
    mapperW.disable(SerializationFeature.WRITE_DATES_AS_TIMESTAMPS);

    mapperW.enable(SerializationFeature.INDENT_OUTPUT);

    File fichier = new File(chemin);
    if (!fichier.exists()){
        fichier.createNewFile();
    }
    mapperW.writerWithDefaultPrettyPrinter().writeValue(new File(chemin), evenementMap);
    System.out.println("Evenment sauvegarder dans " + chemin);
}
```

deserialization

```java
public Map<String, Evenement> charger(String chemin) throws IOException {  1 usage   GAUSS-TPAC
    ObjectMapper mapper= new ObjectMapper();

    mapper.registerSubtypes(Conference.class, Concert.class);
    return mapper.readValue(new File(chemin), new TypeReference<Map<String, Evenement>>() {});   GAUS

}
```

La persistance des données a été assurée par l'implémentation des méthodes de sérialisation et désérialisation JSON, situées dans la classe EvenementSerializer. Grâce à la bibliothèque Jackson, ces méthodes permettent d'enregistrer et de recharger les événements dans un fichier JSON, assurant ainsi la sauvegarde et la restauration du système. L'utilisation des annotations @JsonTypeInfo et @JsonSubTypes dans la classe abstraite Evenement a permis de gérer l'héritage et de garantir la reconstruction correcte des sous-classes lors de la désérialisation. Cette approche a été choisie pour sa simplicité, sa robustesse et sa compatibilité avec les standards actuels.

Notifications asynchrones avec `CompletableFuture`

Afin de simuler un système distribué plus réaliste, l'envoi des notifications a été implémenté de manière asynchrone en utilisant la classe CompletableFuture. Cette approche permet de déclencher les notifications en parallèle, sans bloquer le fil d'exécution principal. Concrètement, la méthode envoyerNotificationAsync encapsule l'appel à envoyerNotification dans un CompletableFuture.runAsync(), avec l'introduction d'un délai aléatoire simulant le temps de transmission réseau. Cela permet d'observer des notifications arrivant dans un ordre non déterministe, reflétant le comportement d'un vrai système distribué. Cette solution démontre l'adaptabilité du projet aux contraintes asynchrones et offre une illustration concrète de la **programmation concurrente** et des **systèmes événementiels** modernes.

```java
@Override 1 usage  GAUSS-TPAC
public void envoyerNotificationAsync(String message) {
    CompletableFuture.runAsync(()->{
        try{
            int delay = ThreadLocalRandom.current().nextInt( origin: 1000, bound: 3000);
            Thread.sleep(delay);
            envoyerNotification(message);
        } catch (InterruptedException e){
            Thread.currentThread().interrupt();
            System.err.println("Notification interrompue: "+ e.getMessage());
        }
    });
}
```

Explication

- **CompletableFuture** : exécute la notification en parallèle pour simuler un envoi différé.
- **ThreadLocalRandom** : génère un délai aléatoire pour la simulation.
- **Pourquoi ?** : rend le système réactif et réaliste, simulant un réseau distribué.

Gestion des exceptions personnalisées

```java
package exception;

public class EvenementDejaExistantException extends RuntimeException {  4 usages  GAUSS-TPAC
    public EvenementDejaExistantException(String message) {  1 usage  GAUSS-TPAC
        super(message);

    }
}
```

Explication:

- **Exception personnalisée** : fournit un message explicite pour des erreurs métier spécifiques.
- **Pourquoi ?** : améliore la clarté, la lisibilité et le débogage du

4- PHASE DE TEST

Les tests unitaires ont été conçus et exécutés avec **JUnit 5**, afin de valider le bon fonctionnement des principales fonctionnalités du système. Un soin particulier a été apporté à la couverture des cas critiques, tels que l'inscription d'un participant, la gestion des exceptions personnalisées, et la sérialisation/désérialisation des événements.

Les tests ont permis de vérifier la levée des exceptions comme CapaciteMaxAtteinteException ou EvenementDejaExistantException lors de tentatives non valides. La bonne réception des notifications, y compris en mode asynchrone, a également été testée. De plus, les opérations de sauvegarde et de chargement ont été validées pour garantir l'intégrité des données persistées.

Ces tests assurent une couverture d'au moins **70%** du code, démontrant la robustesse et la fiabilité du système.

Test d'inscription d'un participant

On fait les testAjouterParticipantDepasseCapaciteMax, testParticipantDejaInscrit, testParticipantValideNombreParticipant

ci dessous pour testParticipantvalideNombreParticipant

```
@Test  GAUSS-TPAC
@DisplayName("test participant valide, nombre participant")
public void testParticipantValideNombreParticipant() {
    Evenement evenement= new Concert( id: "2",  nom: "Concert Test", LocalDateTime.now(),  lieu: "Lyon",  capaciteMax: 5,  artiste: "Artiste XYZ",  genreMusical: "Pop");

    Participant p1= new Participant( id: "p1",  nom: "Alice",  email: "alice@example.com");

    evenement.ajouterParticipant(p1);

    assertTrue(evenement.getParticipants().contains(p1));
    assertEquals( expected: 1, evenement.getParticipants().size());
}
```

Explication

- **Test unitaire JUnit** : vérifie qu'un participant est bien ajouté.
- **Assertions (assertEquals, assertTrue)** : garantissent la validité des résultats.
- **Pourquoi ?** : valide la logique métier d'ajout de participants.

test pour levé les exceptions

testAjouterParticipantDepasseCapacité

```java
@Test  GAUSS-TPAC
@DisplayName("test ajoutParticipant depassant  capacité max de 1")
public void testAjouterParticipantDepasseCapaciteMax() {
    Evenement evenement =new Conference( id: "1", nom: "Conference test", LocalDateTime.now(), lie

    Participant p1 =new Participant( id: "p1", nom: "Alan", email: "alan@gmail.com");
    Participant p2 =new Participant( id: "p2", nom: "lin", email: "lin@gmail.com");

    evenement.ajouterParticipant(p1);

    assertThrows(CapaciteMaxAtteinteException.class, () ->{
        evenement.ajouterParticipant(p2);
    });
}
```

Explication

- **assertThrows** : vérifie qu'une exception spécifique est levée en cas de dépassement de capacité.
- **Pourquoi ?** : valide la robustesse et la fiabilité des contrôles métier.

Test de serialization

```java
@Test  GAUSS-TPAC
@DisplayName("test sauvegarde des evenements")
public void testSauvegardeDesEvenements() {
    String chemin="evenementsTest.json";
    Evenement event =new Conference( id: "1",  nom: "Conférence Test", LocalDateTime.now(),  lieu: "Paris",  capaciteMax: 2,  theme: "IA");
    gestion.ajouterEvenement(event.getId(), event);

    gestion.sauvegarder(chemin);

    File file = new File(chemin);
    assertTrue(file.exists());
    assertTrue( condition: file.length()>0);

    assertTrue(file.delete());
}
```

Explication

- **Test la persistance** : sauvegarde et recharge les données JSON.
- **Assertions** : vérifient que les données sont cohérentes avant/après.
- **Pourquoi ?** : garantit l'intégrité et la fiabilité de la persistance.

5- PERSPECTIVE ET AMELIORATION

Bien que le système soit fonctionnel et complet dans sa version actuelle, plusieurs améliorations pourraient être envisagées :

Intégration d’une **base de données relationnelle** pour la persistance, afin de garantir la scalabilité et la robustesse.

Ajout de **nouvelles fonctionnalités**, telles que la recherche avancée d’événements, la gestion des disponibilités des participants, ou encore l’intégration d’un système de messagerie externe (email, SMS).

Renforcement de la **gestion concurrente**, notamment pour gérer de nombreux utilisateurs simultanés et garantir la cohérence des données.

Adoption d’une **approche orientée microservices** et déploiement dans un environnement distribué, pour transformer l’application en une solution industrielle.

Ces pistes ouvrent la voie à une évolution du système vers un produit plus abouti, prêt à répondre aux besoins réels d’un environnement professionnel.

6- CONCLUSION

Ce projet a permis de concevoir et développer un système complet et robuste de gestion d'événements, en appliquant les concepts avancés de la **programmation orientée objet** et les **design patterns** tels que Singleton et Observer. L'intégration de la sérialisation JSON avec **Jackson**, ainsi que l'utilisation de **JUnit** pour les tests unitaires, ont contribué à la qualité et à la fiabilité de l'application.

L'implémentation des **notifications asynchrones** avec CompletableFuture a enrichi le système en apportant une dimension réaliste et professionnelle à la gestion des événements. Le projet a permis de renforcer les compétences en conception logicielle, en manipulation de collections, et en gestion d'exceptions métier.

En résumé, ce système modulaire et extensible démontre une approche professionnelle et évolutive de la programmation orientée objet, tout en mettant l'accent sur la fiabilité et la performance.