École Nationale Supérieure Polytechnique de Yaoundé
National Advanced School of Engineering of Yaounde

Département de Génie Informatique
Departement of Computer Engineering

U.E : PROGRAMMATION ORRIENTÉE OBJET JAVA 2
U.E : OBJECT ORIENTED PROGRAMMING JAVA 2

Event Management

Application de Système de Gestion d'Événements Distribué

Rapport de Projet

Réalisé par l'étudiant :
Leonel Delmat AZANGUE 22P206

Niveau : III

Sous la supervision de :

Dr KUNGNE

Année académique : 2024-2025

Table des matières

0.1 Introduction

L'objectif de ce projet est de concevoir et développer une application JavaFX permettant aux organisateurs de gérer différents types d'événements (concerts, conférences, etc.) avec des fonctionnalités riches : création, modification, suivi, annulation, visualisation, etc. Le système est interactif, modulaire et facilement extensible.

0.2 Contexte, Problématique et Objectifs

0.2.1 Contexte

Dans le cadre du cours de Programmation Orientée Objet (POO) en Java, un projet a été proposé afin d'appliquer les principes avancés de la conception logicielle, en particulier dans un contexte distribué. Le projet consiste à développer une application de gestion d'événements à destination de différentes catégories d'utilisateurs (organisateurs, participants), avec des fonctionnalités telles que la création, la participation, la visualisation, la communication et les notifications. Ce projet est réalisé dans le cadre académique de l'École Nationale Supérieure Polytechnique de Yaoundé (ENSPY), et illustre une mise en pratique des acquis en POO, JavaFX, design patterns, architecture logicielle, et tests unitaires.

0.2.2 Problématique

Dans les milieux académiques et professionnels, la gestion manuelle des événements (séminaires, conférences, réunions, ateliers, etc.) présente souvent des limites : pertes d'informations, manque de coordination, absence de rappels ou de notifications automatiques, et difficulté d'accès aux données en temps réel. De plus, avec la multiplicité des rôles et la distribution géographique des utilisateurs, une solution centralisée, ergonomique, et évolutive devient nécessaire.

0.2.3 Objectifs

L'objectif principal de ce projet est de concevoir et développer une application Java orientée objet, robuste, intuitive et conforme aux bonnes pratiques de conception logicielle, permettant :

— Aux organisateurs de créer, modifier, annuler et suivre les événements.

— Aux participants de consulter, s'inscrire, dés'inscrire, être notifiés et interagir.

— De garantir une architecture modulaire facilitant la maintenance et l'évolution.

— D'utiliser des design patterns pertinents (Observer, Singleton, MVC, etc.).

— D'intégrer des tests unitaires pour assurer la qualité logicielle.

— De proposer une interface graphique conviviale via JavaFX.

0.3 Architecture et Diagrammes UML

L'architecture de l'application repose sur le pattern MVC (Modèle-Vue-Contrôleur). Les composants sont organisés par packages :

— `model` : classes métier (Evenement, Organisateur, etc.)
— `controller` : logique des interfaces JavaFX
— `persistence` : sérialisation JSON/XML
— `utils` : outils (transition de vues, preview fichiers)
— `service` : services de gestion des différents entités
— `exceptions` : exceptions personnalisées
— `main` : point d'entrée de l'application
— `view` : fichiers FXML pour les interfaces utilisateur
— `test` : tests unitaires
— `resources` : ressources statiques (images, styles CSS)

0.3.1 Diagramme de Contexte

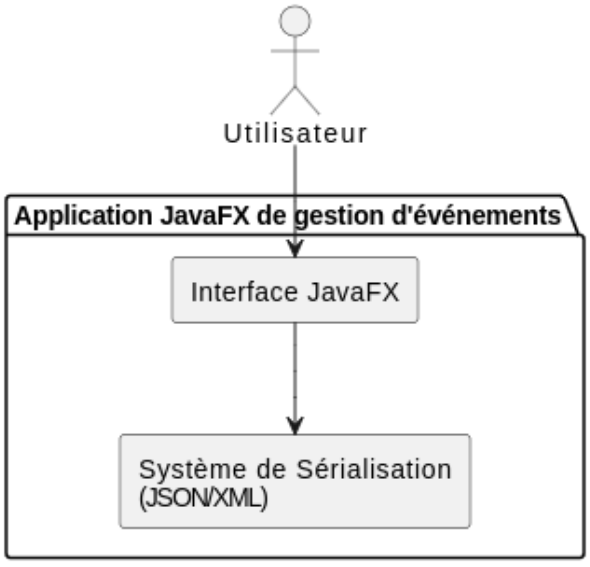


FIGURE 1 – Diagramme de contexte de l'application

0.3.2 Diagramme de Packages

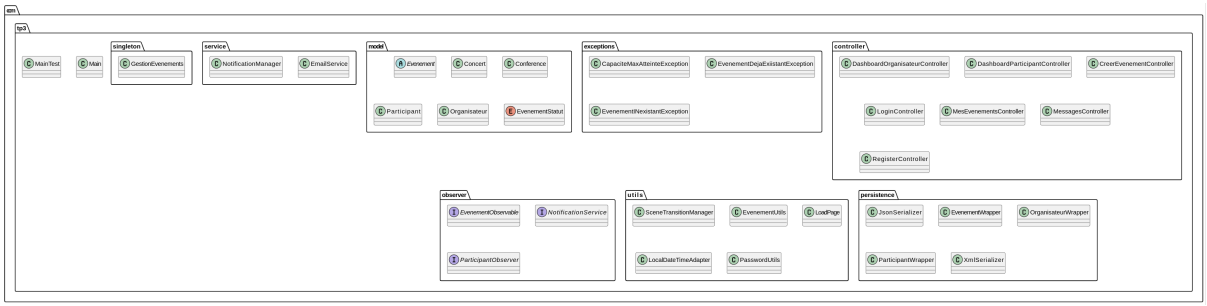


FIGURE 2 – Diagramme de packages de l'application

0.3.3 Diagramme de Classes Métier

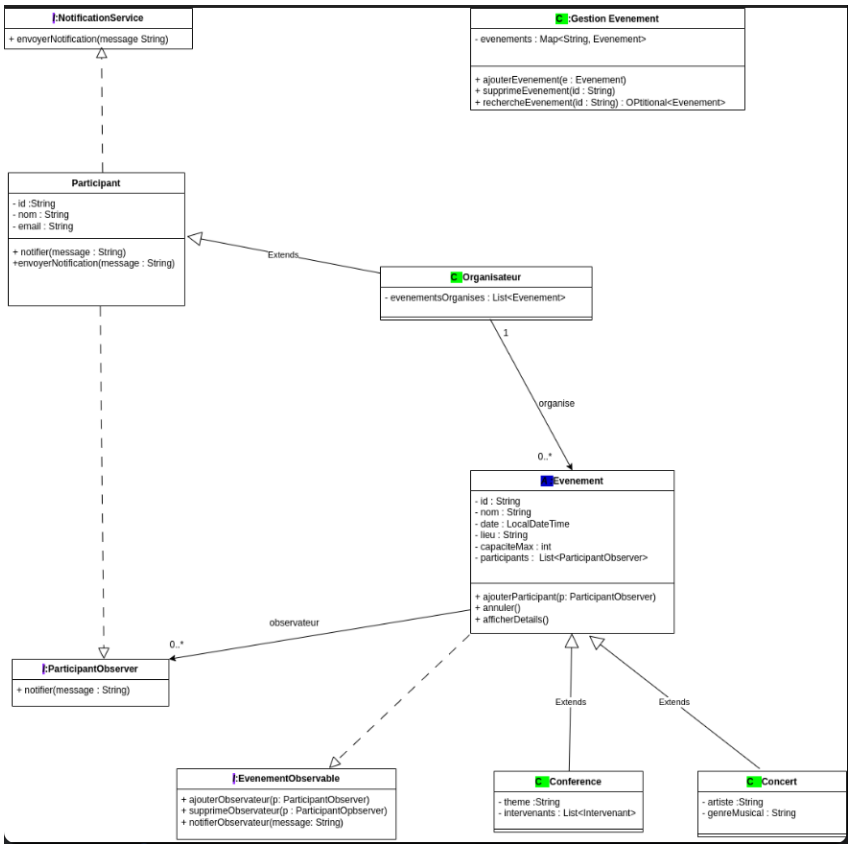


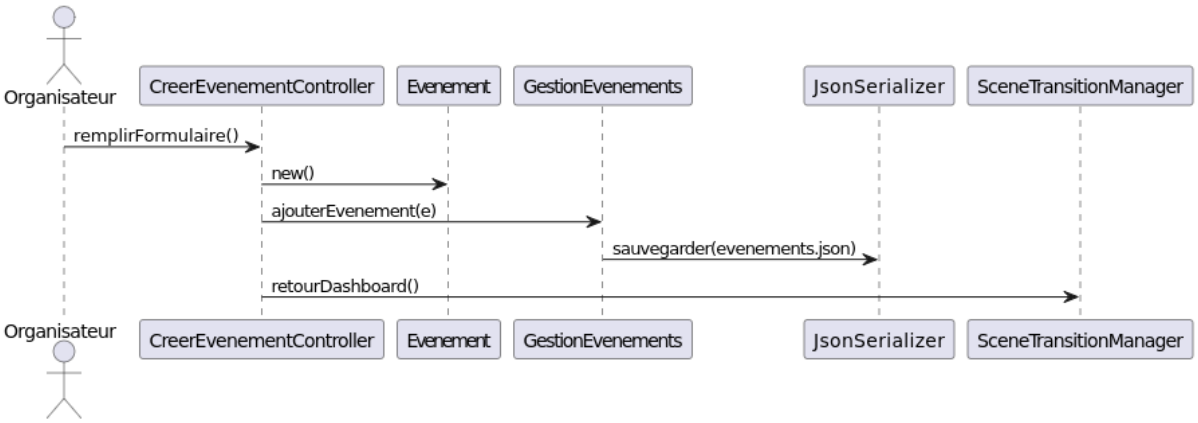
FIGURE 3 – Diagramme de classes métier de l'application

0.3.4 Diagramme de Cas d'Utilisation

0.3.5 Diagrammes de Séquence

— Séquence système : création d'un événement

— Séquence technique : persistance via JSON



0.3.6 Diagramme d'États

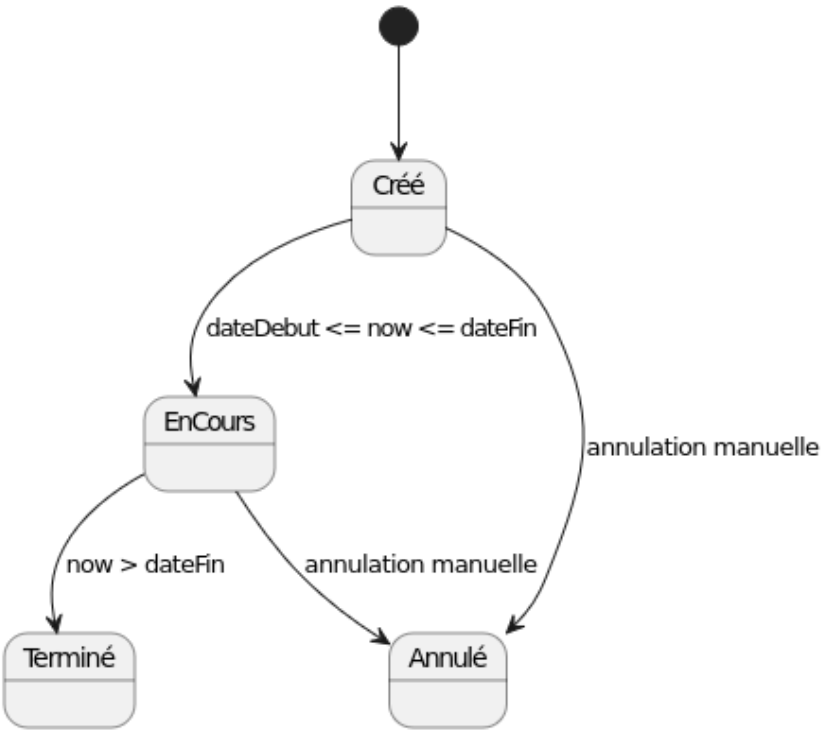


FIGURE 4 – Diagramme d'états des événements

0.4 Design Patterns Utilisés

0.4.1 Singleton

Utilisé pour la classe `GestionEvenements` afin de centraliser les opérations sur les événements et garantir une instance unique.

0.4.2 Strategy

Les classes `JsonSerializer` et `XmlSerializer` permettent de changer dynamiquement la méthode de persistance.

0.4.3 MVC

Adopté pour séparer clairement la logique métier, la vue et le contrôleur.

0.5 Exemples de Code Importants

0.5.1 Sérialisation et Deserialization avec JSON

```
1  public class JsonSerializer {
2
3      // ObjectMapper de Jackson pour la sérialisation/dése
       rialisation
4      private static final ObjectMapper mapper = new ObjectMapper();
5
6      static {
7          // Support des types Java 8 (LocalDate, LocalDateTime, etc
               .)
8          mapper.registerModule(new JavaTimeModule());
9
10         // Active le formatage lisible (joli indenté)
11         mapper.enable(SerializationFeature.INDENT_OUTPUT);
12         // Utilise le format ISO pour les dates (ex: "2025-05-26
               T14:30:00")
13         mapper.disable(SerializationFeature.
               WRITE_DATES_AS_TIMESTAMPS);
14         // Ignore les propriétés inconnues dans le JSON lors du
               chargement
15         mapper.configure(DeserializationFeature.
               FAIL_ON_UNKNOWN_PROPERTIES, false);
16     }
17
18     /**
19      * Sauvegarde tous les événements dans un fichier JSON.
20      *
21      * @param gestion  L'instance de GestionEvenements contenant
            les données
22      * @param filename Le nom du fichier (ex : "evenements.json")
```

```
23      * @throws IOException En cas de problème d'écriture
24      */
25     public static void saveEvenementToJson(GestionEvenements
           gestion, String filename) throws IOException {
26         File file = new File(filename);
27 
28         // Création du dossier parent s'il n'existe pas
29         File parentDir = file.getParentFile();
30         if (parentDir != null && !parentDir.exists()) {
31             parentDir.mkdirs();
32         }
33 
34         // Création du fichier s'il n'existe pas
35         if (!file.exists()) {
36             file.createNewFile();
37         }
38 
39         // Écriture des événements au format JSON
40         mapper.writeValue(file, gestion.getEvenements());
41     }
42 
43     /**
44      * Charge les événements depuis un fichier JSON et les insère
           dans la gestion.
45      *
46      * @param gestion  L'objet qui contiendra les événements charg
           és
47      * @param filename Le fichier JSON source
48      * @return La map d'événements chargés
49      * @throws IOException En cas de lecture invalide
50      */
51     public static Map<String, Evenement> loadEvenementFromJson(
           GestionEvenements gestion, String filename) throws
           IOException {
52         // Lecture du fichier JSON en tant que Map<String,
               Evenement>
53         Map<String, Evenement> loaded = mapper.readValue(
54             new File(filename),
55             mapper.getTypeFactory().constructMapType(
56                 java.util.HashMap.class, String.class, Evenement.
                       class
57             )
58         );
59 
60         // Injection dans l'instance de gestion
61         gestion.setEvenements(loaded);
62         return loaded;
63     }
64 }
```

Listing 1 – Classe JsonSerializer pour gérer les événements

0.5.2 GestionEvenements

```java
public class GestionEvenements {

    // Instance unique (singleton)
    private static GestionEvenements instance;

    // Stockage des événements en mémoire (cache)
    private Map<String, Evenement> evenements = new HashMap<>();

    // Logger pour journalisation
    private final Logger LOGGER = Logger.getLogger(
       GestionEvenements.class.getName());

    // Constructeur prive pour empecher l'instanciation directe (
       pattern Singleton)
    private GestionEvenements() {
        try {
            // Chargement des evenements depuis le fichier JSON au
                demarrage
            Map<String, Evenement> loaded = JsonSerializer.
               loadEvenementFromJson(this, "evenements.json");
            if (loaded != null) {
                this.evenements = loaded;
            }
        } catch (IOException e) {
            LOGGER.warning(" Aucune donnee JSON chargee au
               demarrage : " + e.getMessage());
        }
    }
    // Méthode statique pour obtenir l'instance unique de la
       classe
    public static GestionEvenements getInstance() {
        if (instance == null) {
            instance = new GestionEvenements();
        }
        return instance;
    }
    /**
     * Ajoute un événement dans la liste.
     * @param event L'evenement a ajouter
     * @throws EvenementDejaExistantException si un evenement avec
         le meme ID existe deja
     */
    public void ajouterEvenement(Evenement event) throws
       EvenementDejaExistantException {
        // Vérifie si un événement avec cet ID existe déjà
        if (evenements.containsKey(event.getId())) {
            throw new EvenementDejaExistantException(" Événement d
               éjà existant avec l'ID : " + event.getId());
        }
```

```
41
42          // Ajoute l'evenement à la map
43          evenements.put(event.getId(), event);
44          LOGGER.info(" AJOUT : " + event.getNom());
45
46          // Sauvegarde dans les fichiers JSON et XML
47          sauvegarder();
48      }
49  }
```

Listing 2 – Classe JsonSerializer pour gérer les événements

0.6 Captures d'Écran

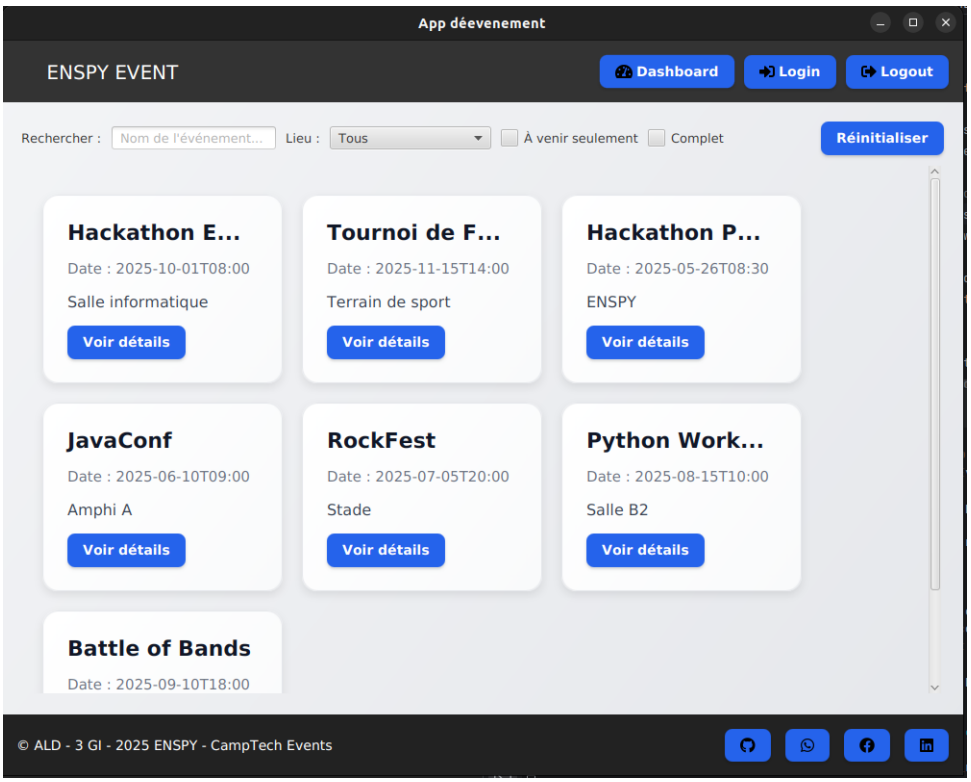


FIGURE 5 – Vue Acceuil

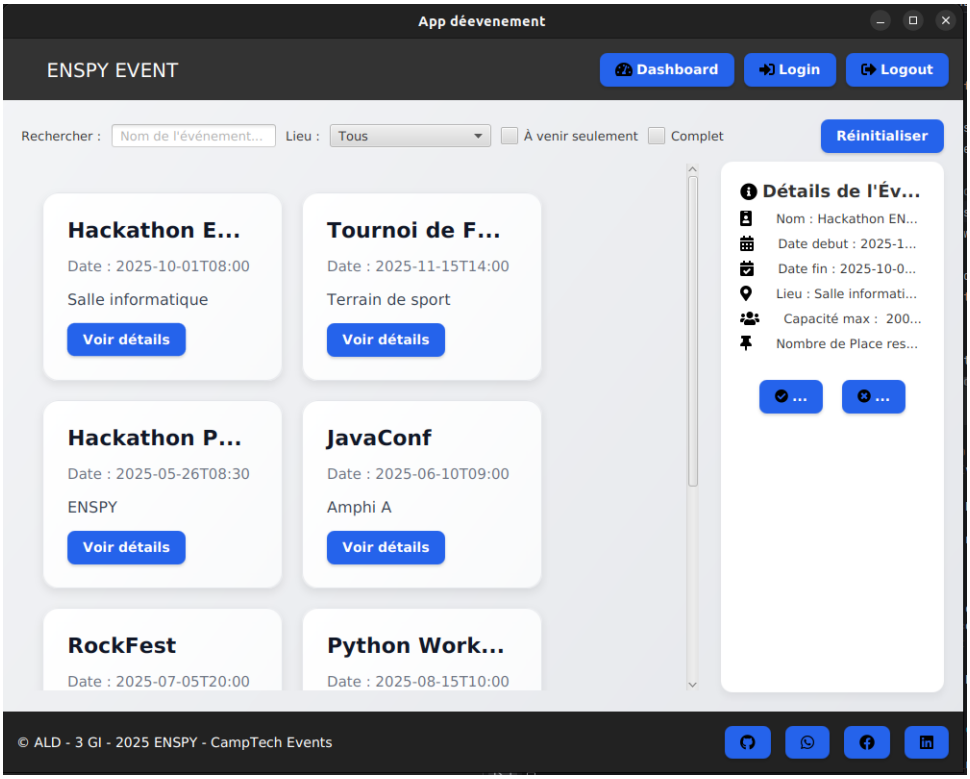


FIGURE 6 – Vue Acceuil 2

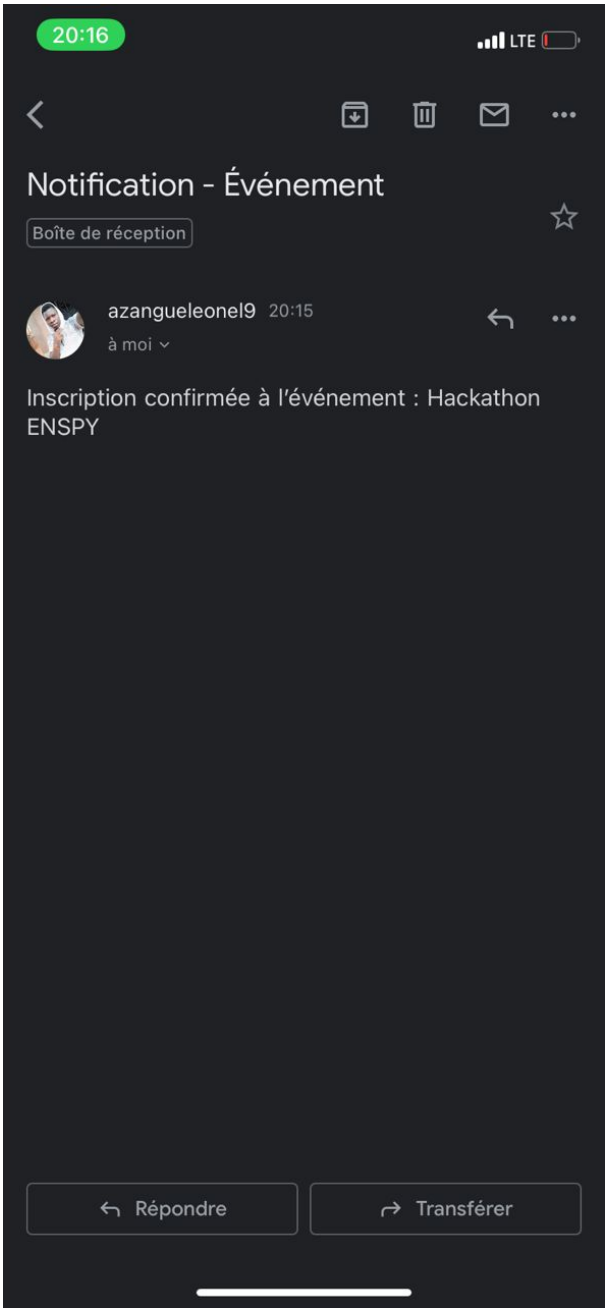


FIGURE 7 – Reception de notification email

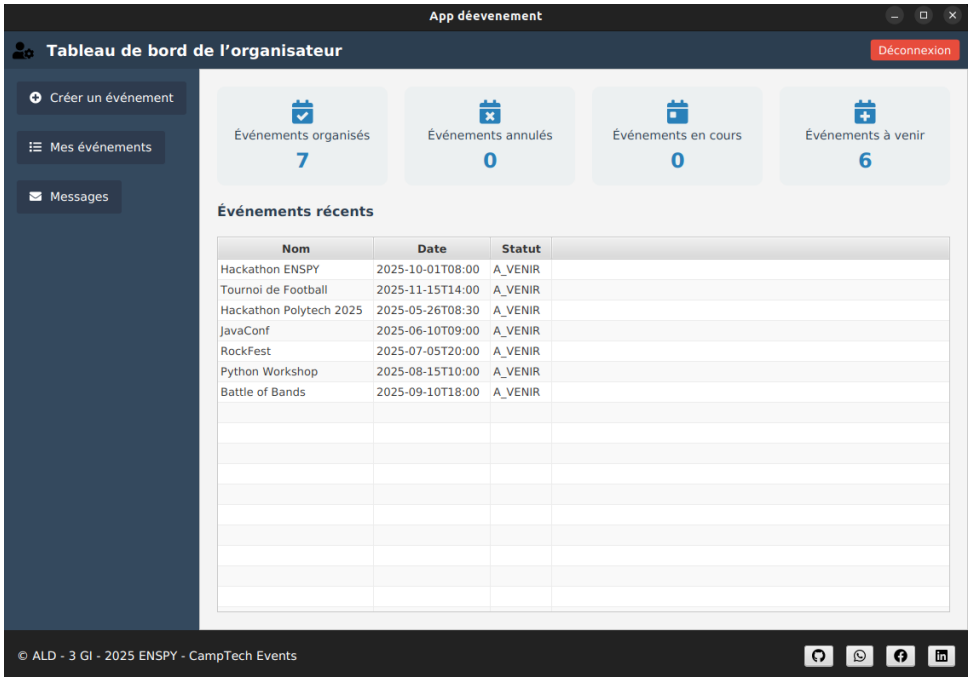


FIGURE 8 – Vue tableau de bord de l'organisateur

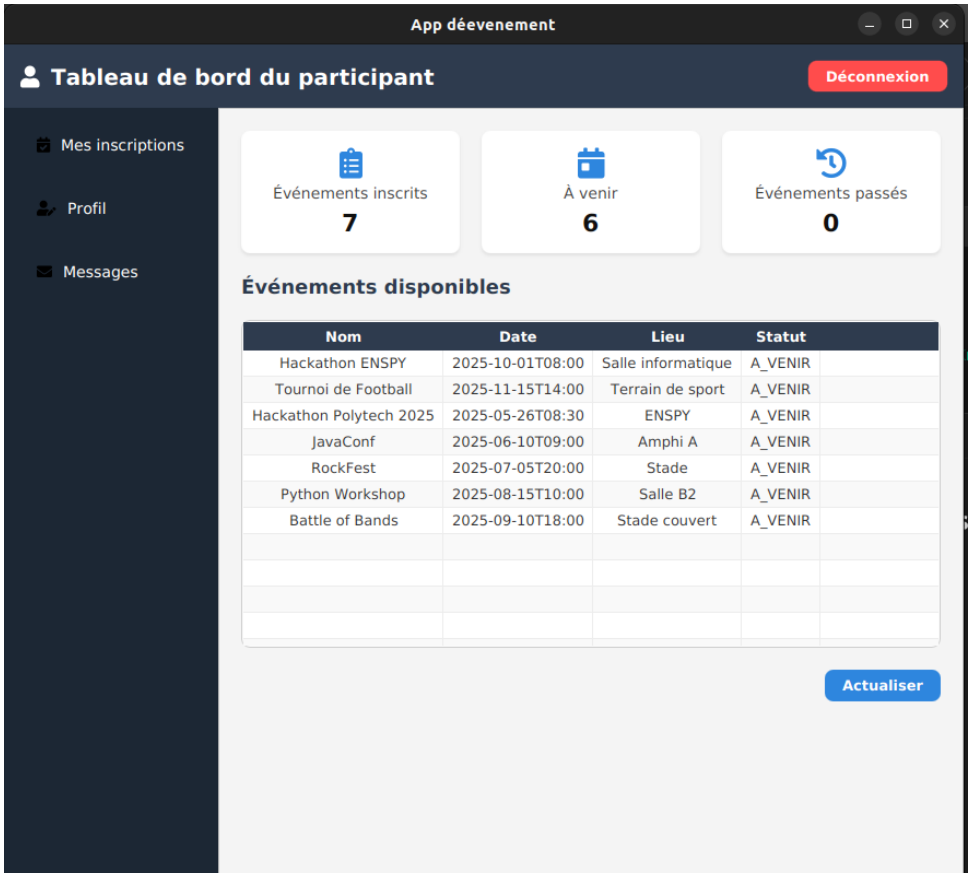


FIGURE 9 – Vue tableau de bord des participants

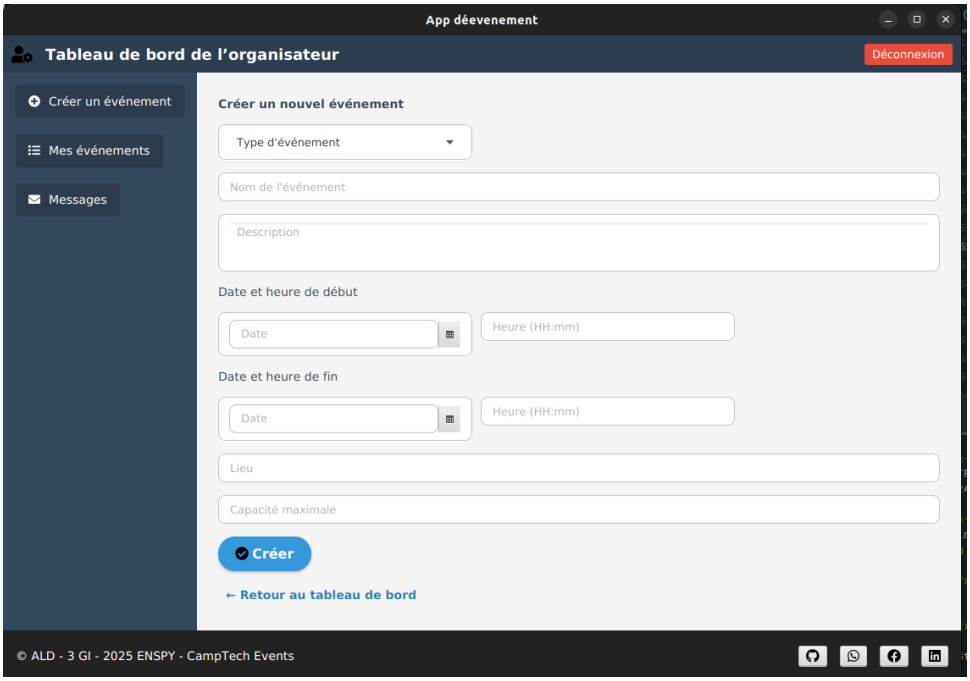


FIGURE 10 – Formulaire de création d'un événement

0.7 Résultats de la sérialisation JSON

Voici un extrait du fichier JSON généré contenant les événements :

```
{
  "E001" : {
    "id" : "E001",
    "nom" : "JavaConf",
    "dateDebut" : "2025-06-10T09:00:00",
    "dateFin" : "2025-07-05T20:00:00",
    "lieu" : "Amphi A",
    "capaciteMax" : 100,
    "statut" : "A_VENIR",
    "theme" : "POO avancée",
    "intervenants" : [ "Prof. Martin" ],
    "type" : "conference",
    "nombreParticipants" : 0
  },
  "E002" : {
    "id" : "E002",
    "nom" : "RockFest",
    "dateDebut" : "2025-07-05T20:00:00",
    "dateFin" : "2025-07-05T22:00:00",
    "lieu" : "Stade",
    "capaciteMax" : 5000,
    "statut" : "A_VENIR",
    "artiste" : "The Rockers",
    "genreMusical" : "Rock",
    "type" : "concert",
    "nombreParticipants" : 0
```

```
27    }
28 }
```

0.8 Résultats des Tests Unitaires

— Test de création d'un événement : `OK`

— Test de sérialisation JSON : `OK`

— Test de transitions d'écrans : `OK`

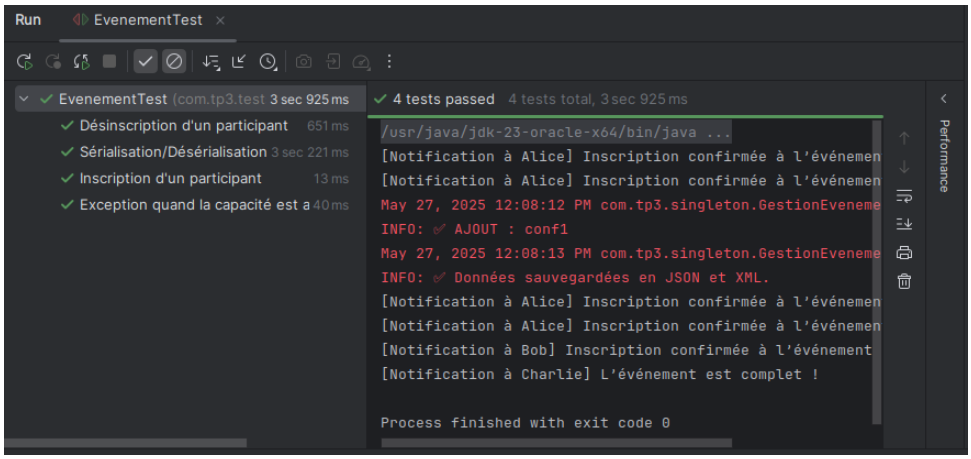


FIGURE 11 – Resultat des tests avec JUnit

0.9 Conclusion

Limites actuelles :

— Absence d'une base de données relationnelle pour la persistance des événements, utilisateurs et messages.

— Communication limitée à l'environnement local (pas de déploiement réseau ou cloud).

— Sécurité de l'application minimale : pas d'authentification forte ni de gestion des sessions utilisateurs.

— Couplage relativement fort entre les composants métier et les vues JavaFX.

Perspectives d'évolution :

— Intégration avec une API REST (via Spring Boot ou Django) pour permettre une architecture distribuée et multiplateforme.

— Mise en place d'un système d'authentification avancé avec rôles (organisateur, participant, modérateur, etc.).

— Ajout d'une base de données relationnelle (PostgreSQL, MySQL) pour stocker les événements, utilisateurs et messages de manière pérenne.

— Gestion des notifications en temps réel (WebSockets, Firebase).

— Possibilité de gérer les inscriptions des participants via un portail en ligne.

— Amélioration de l'expérience utilisateur avec un design responsive et une interface web complémentaire.

En somme, cette application constitue une base solide pour un système de gestion d'événements distribués, évolutif et adaptable à des contextes académiques, associatifs ou professionnels.

Annexe – Démarche de Réalisation du TP Final POO

ÉTAPE 1 – Analyse & Compréhension du sujet

- Lire soigneusement le sujet : Relever les concepts demandés :
  - Héritage, Polymorphisme, Interfaces
  - Design Patterns (Observer, Singleton, Factory, Strategy)
  - Exceptions personnalisées
  - Collections (Map, List)
  - Sérialisation JSON/XML
  - Programmation événementielle & asynchrone
  - Tests JUnit (70% de couverture)
- Identifier les entités principales : Evenement, Conference, Concert, Participant, Organisateur

ÉTAPE 2 – Modélisation UML

- Créer les classes selon le diagramme demandé (Evenement, Conference, Concert, etc.)
- Définir les relations UML (Héritage, Association, Observer, Singleton)
- Utiliser draw.io, StarUML ou version manuscrite

ÉTAPE 3 – Implémentation des classes de base

- Classe abstraite Evenement avec méthodes `ajouterParticipant()`, `annuler()`, etc.
- Sous-classes Conference et Concert
- Singleton `GestionEvenements` avec gestion CRUD

ÉTAPE 4 – Implémenter le pattern Observer

- Créer `EvenementObservable` et `ParticipantObserver`
- Gérer les notifications lors des changements d'état

ÉTAPE 5 – Exceptions personnalisées

- Créer et utiliser : `CapaciteMaxAtteinteException`, `EvenementDejaExistantException`, etc.
- Utiliser les blocs `try/catch` dans les fonctions critiques

ÉTAPE 6 – Sérialisation JSON/XML

- Utiliser Jackson (JSON) et JAXB (XML)
- Ajouter les méthodes : `sauvegarderVersFichierJSON()`, `chargerDepuisFichierJSON()`

ÉTAPE 7 – Streams & Lambdas

— Utiliser les Streams pour filtrer, trier, chercher des événements

ÉTAPE 8 – Programmation Asynchrone

— Utiliser `CompletableFuture.runAsync()` pour simuler des notifications différées

ÉTAPE 9 – Tests Unitaires JUnit

— Tester les inscriptions, annulations, exceptions

— Atteindre 70% de couverture avec JaCoCo

ÉTAPE 10 – Rapport PDF

— Structure : Page de garde, Introduction, UML, Patterns, Code, Captures, Résultats, Conclusion

ÉTAPE 11 – Dépôt GitHub

— Créer un dépôt GitHub

— Ajouter le code source, tests, README

— Inviter le correcteur

ÉTAPE 12 – Présentation orale

— Préparer un support (5–10 min)

— Être prêt à justifier les choix techniques

Récapitulatif des livrables

— Code source commenté

— Tests unitaires $\geq 70\%$

— Rapport PDF

— Dépôt GitHub

— Présentation orale

0.10 Références

— Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

— Oracle. *The Java™ Tutorials*. Disponible sur https://docs.oracle.com/javase/tutorial/

— Oracle. *JavaFX Documentation*. Disponible sur https://openjfx.io

— Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head First Design Patterns*. O'Reilly Media.

— Martin, R. C. (2008). *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall.

— Mackenzie, C. (2019). *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley.

— Documentation officielle de JUnit : https://junit.org/junit5/

— draw.io. *Outil de modélisation UML en ligne*. Disponible sur https://draw.io