Table des matières

1 Introduction

Ce rapport analyse l'application de gestion d'événements développée en Java, mettant en évidence son architecture orientée objet, ses patterns de conception, ses fonctionnalités, ses mécanismes de synchronisation, son interface utilisateur moderne basée sur JavaFX, et ses capacités de sérialisation. L'application permet la gestion complète des événements, des participants et des organisateurs, avec des interfaces dédiées pour différents rôles et une synchronisation en temps réel des données.

2 Cahier des Charges

2.1 Besoins Fonctionnels

1. **Gestion des Événements** :
   — Création, modification et suppression d'événements (conférences, concerts).
   — Recherche et filtrage par lieu, date ou type.
   — Gestion de la capacité maximale et validation d'unicité.
   — Annulation d'événements avec notifications automatiques.
2. **Gestion des Participants** :
   — Inscription et désinscription avec vérification de capacité.
   — Notifications automatiques pour les changements (inscription, annulation).
   — Support pour différents rôles (participants, organisateurs, intervenants).
3. **Interface Utilisateur** :
   — Interfaces spécifiques pour administrateurs, organisateurs et participants.
   — Affichage des statistiques en temps réel (événements, participants, inscriptions).
   — Notifications modernes (alertes, toasts, snackbars, dialogues de progression).
   — Design responsive avec Material Design et police Poppins.
4. **Synchronisation et Persistance** :
   — Synchronisation automatique avec sauvegarde périodique en JSON et XML.
   — Exportation des données avec métadonnées (timestamp, nombre d'éléments).
   — Initialisation de données de démonstration pour le développement.
5. **Gestion des Erreurs** :
   — Validation des données d'entrée.
   — Exceptions métier pour capacité, unicité et erreurs de participants.
   — Affichage d'erreurs via alertes stylisées.

2.2 Spécifications Techniques

— **Langage** : Java 17 avec Streams, `LocalDateTime`, collections génériques.
— **Interface Utilisateur** : JavaFX avec CSS personnalisé (Material Design).
— **Patterns de Conception** :
   — Singleton pour `GestionEvenements` et `DataSynchronizer`.
   — Observer pour notifications en temps réel.
   — Template Method pour hiérarchie des événements.
   — Strategy pour services de notification.
— **Sérialisation** : Jackson pour JSON, API DOM pour XML.
— **Synchronisation** : `ScheduledExecutorService` pour sauvegarde automatique, thread-safety via double-checked locking.
— **Police** : Poppins, avec fallback sur police par défaut.
— **Libraries** : JavaFX, Jackson, API DOM.
— **Environnement** : Compatible JVM, sans dépendances externes lourdes.

3 Vue d'ensemble de l'Architecture

3.1 Structure du Projet

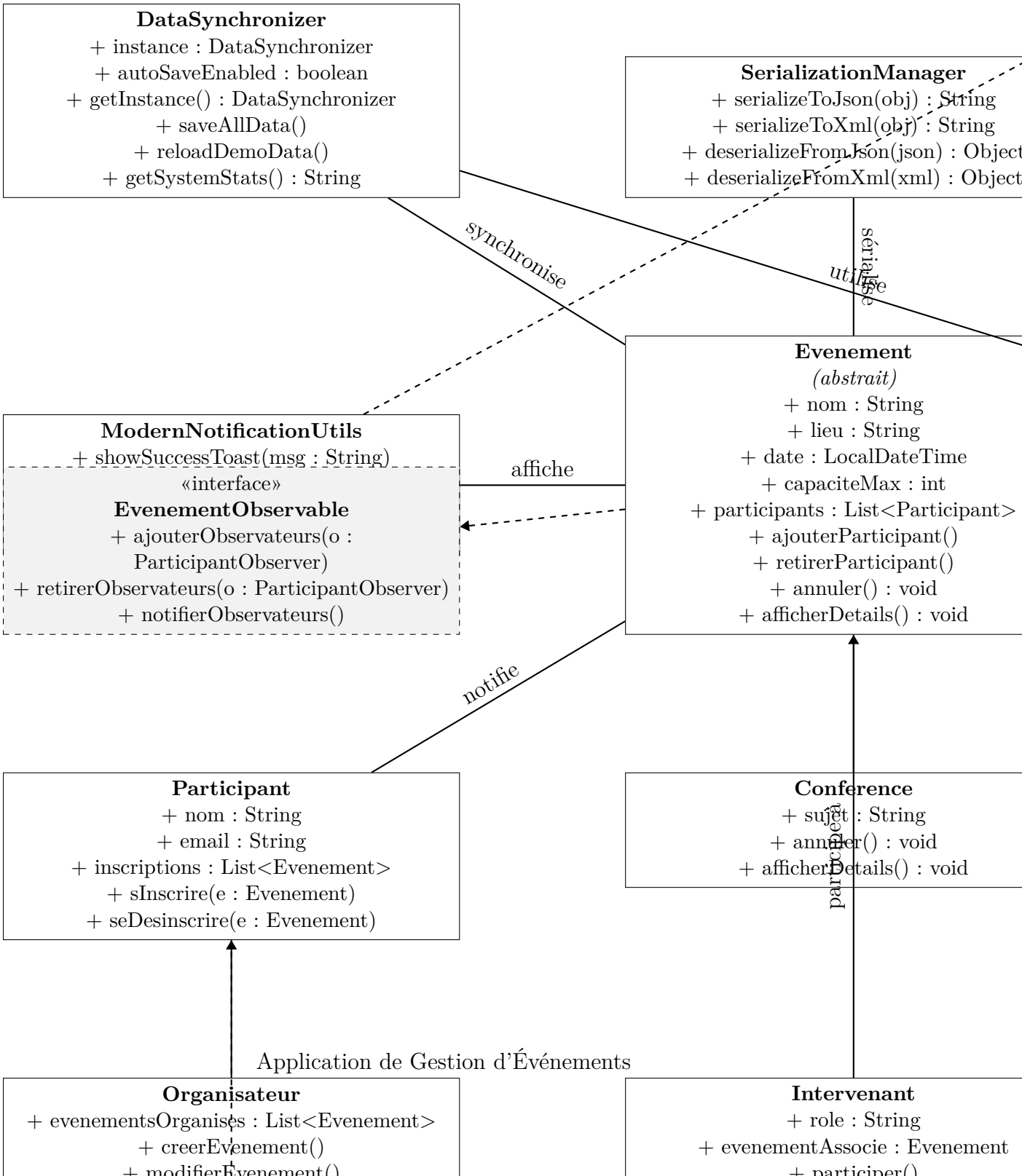
L'application est organisée en packages :

— `com.gestion.evenements` : Classes principales et lanceur.
— `com.gestion.evenements.model` : Modèles de données.
— `com.gestion.evenements.observer` : Pattern Observer.
— `com.gestion.evenements.exception` : Exceptions métier.
— `com.gestion.evenements.model.evenementparticulier` : Types d'événements.
— `com.gestion.evenements.model.notification` : Notifications.
— `com.gestion.evenements.util` : Synchronisation.
— `com.gestion.evenements.ui` : Interface utilisateur.
— `com.gestion.evenements.ui.utils` : Utilitaires UI.
— `com.gestion.evenements.serialization` : Sérialisation.

3.2 Classes Principales

1. `Evenement` : Classe abstraite pour événements.
2. `GestionEvenements` : Gestionnaire principal (Singleton).
3. `Participant` : Représentation des participants.
4. `Organisateur` : Extension de `Participant`.
5. `Intervenant` : Intervenants des événements.
6. `DataSynchronizer` : Synchronisation des données.
7. `ModernNotificationUtils` : Notifications UI.
8. `SerializationManager` : Sérialisation JSON/XML.
9. `ApplicationLauncher` : Lanceur JavaFX.

3.3 Diagramme de Classes



4 Analyse des Patterns de Conception

4.1 Pattern Singleton

`GestionEvenements` et `DataSynchronizer` implémentent le Singleton thread-safe :

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

Listing 1 – Singleton dans GestionEvenements

Avantages :

— Instance unique pour gestion centralisée.

— Thread-safety via `synchronized` ou double-checked locking.

— Centralisation des événements et synchronisation.

4.2 Pattern Observer

Le pattern Observer gère les notifications via `EvenementObservable` et `ParticipantObserver` :

```
public interface EvenementObservable {
    void ajouterObservateurs(ParticipantObserver observer);
    void retirerObservateurs(ParticipantObserver observer);
    void notifierObservateurs(String message);
}
```

Listing 2 – Interface EvenementObservable

Fonctionnement :

— Événements notifient participants et UI (`UIObserver`).

— Découplage entre producteurs et consommateurs.

— `Platform.runLater` pour compatibilité JavaFX.

4.3 Pattern Template Method

`Evenement` utilise Template Method pour `annuler()` et `afficherDetails()`, implémentés par `Conference` et `Concert`.

4.4 Pattern Strategy

Les notifications utilisent Strategy via `NotificationService` :

— Implémentation : `EmailNotificationService`.

— Extensible pour d'autres stratégies.

5 Analyse des Fonctionnalités

5.1 Gestion des Événements

— **Création/Suppression** : Validation d'unicité, nettoyage automatique.

— **Recherche/Filtrage** : Par lieu ou événements futurs.

```
public List<Evenement> rechercherParLieu(String lieu) {
    return evenements.values().stream()
            .filter(e -> e.getLieu().toLowerCase().contains(lieu.
               toLowerCase()))
            .collect(Collectors.toList());
}

public List<Evenement> getEvenementsFuturs() {
    return evenements.values().stream()
            .filter(e -> e.getDate().isAfter(LocalDateTime.now()))
            .sorted(Comparator.comparing(Evenement::getDate))
            .collect(Collectors.toList());
}
```

Listing 3 – Recherche d'événements

5.2 Gestion des Participants

— Inscription avec vérification de capacité.

— Notifications automatiques.

— Hiérarchie : `Participant`, `Organisateur`, `Intervenant`.

5.3 Lanceur d'Application

`ApplicationLauncher` initialise l'application JavaFX :

— Sélection d'interface (administrateur, organisateur, participant) via cartes interactives.

— Statistiques en temps réel via `DataSynchronizer`.

— Interface moderne avec Material Design, police Poppins, animations.

```
private void launchAdminInterface() {
    primaryStage.close();
    GestionEvenementsApp adminApp = new GestionEvenementsApp();
    Stage adminStage = new Stage();
    adminApp.start(adminStage);
}
```

Listing 4 – Lancement d'interface

6 Gestion des Erreurs

6.1 Exceptions Métier

— `CapaciteMaxAtteinteException`.

— `EvenementDejaExistantException`.

— `ParticipantNonTrouveException`.

```
public void ajouterParticipant(Participant participant)
        throws CapaciteMaxAtteinteException {
    if (participants.size() >= capaciteMax) {
        throw new CapaciteMaxAtteinteException(
            "Capacité maximale atteinte pour l'événement " + nom);
    }
    participants.add(participant);
    notifierObservateurs("Nouveau participant ajouté: " + participant.
        getNom());
}
```

Listing 5 – Gestion d'erreur

6.2 Gestion Robuste

Alertes stylisées via `ModernNotificationUtils` pour erreurs utilisateur.

7 Sérialisation et Persistance

7.1 JSON

Jackson pour sérialisation polymorphe :

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

Listing 6 – Configuration Jackson

7.2 XML

`SerializationManager` utilise API DOM :

```
private static Element createEvenementElement(Document doc, Evenement
    evenement) {
    Element evenementElement = doc.createElement("evenement");
    evenementElement.setAttribute("id", evenement.getId());
    evenementElement.setAttribute("type", getEvenementType(evenement));
    // ...
}
```

Listing 7 – Sérialisation XML

8 Programmation Moderne Java

— Streams pour filtrage/transformation.
— `LocalDateTime` pour dates.
— Collections génériques.
— `CompletableFuture` (importé, non utilisé dans extraits).

9 Synchronisation des Données

`DataSynchronizer` gère :
— Sauvegarde automatique via `ScheduledExecutorService`.
— Notifications globales.
— Données de démonstration.

```
private void startAutoSave() {
    scheduledExecutor.scheduleAtFixedRate(
        this::saveAllData,
        autoSaveIntervalMinutes,
        autoSaveIntervalMinutes,
        TimeUnit.MINUTES
    );
}
```

Listing 8 – Sauvegarde automatique

10 Interface Utilisateur

10.1 ModernNotificationUtils

Composants JavaFX :
— Alertes, toasts, snackbars, dialogues de progression.
— Thème Material Design via `modernStyle.css`.

```
public static void showSuccessToast(String message) {
    showToast(message, ToastType.SUCCESS, " ");
}
```

Listing 9 – Notification toast

10.2 ApplicationLauncher

— Sélection d'interface avec cartes interactives et tooltips.
— Statistiques en temps réel.
— Responsive avec animations, mode sombre.

10.3 UIObserver

Relie modèle et UI :

```
public class UIObserver implements ParticipantObserver {
    @Override
    public void notifier(String message) {
        Platform.runLater(() -> {
            if (statusLabel != null) {
                statusLabel.setText(message);
            }
        });
    }
}
```

Listing 10 – UIObserver

11 Points Forts

1. **Extensibilité** : Modularité pour nouveaux événements.
2. **Maintenabilité** : Séparation des responsabilités.
3. **Réutilisabilité** : Patterns découplés.
4. **Robustesse** : Gestion d'erreurs.
5. **Modernité** : JavaFX, Material Design.
6. **Synchronisation** : Sauvegarde automatique.

11.1 Respect des Principes SOLID

— **S** : Responsabilité unique.
— **O** : Extensible sans modification.
— **L** : Substitution de Liskov.
— **I** : Interfaces spécifiques.
— **D** : Dépendance vers abstractions.

12 Recommandations

12.1 Améliorations Techniques

1. **Logging** : SLF4J ou Log4j.
2. **Tests** : JUnit, Mockito.
3. **Configuration** : Externaliser paramètres.
4. **Validation** : Bean Validation.
5. **Documentation** : JavaDoc, guides utilisateur.

12.2 Améliorations Fonctionnelles

1. **Persistance** : Base de données relationnelle.
2. **API REST** : Intégration externe.
3. **Sécurité** : OAuth2.
4. **Performance** : Indexation pour gros volumes.
5. **Accessibilité** : Conformité WCAG.

13 Analyse et Tests du Système

Ce rapport analyse le système de gestion d'événements EventPro, basé sur les fichiers fournis (`GestionEvenementsApp.java`, `LoginView.java`, `WelcomeView.java`, `RegisterView.java`, et `SystemeGestionEvenementsTest.java`). Il couvre l'architecture, les fonctionnalités, les tests unitaires, les points forts, les points à améliorer, et des recommandations pour optimiser le système.

13.1 Architecture du Système

13.1.1 Structure Générale

EventPro est une application JavaFX conçue avec une architecture MVC (Modèle-Vue-Contrôleur) et une synchronisation en temps réel via le pattern Observer. Les principaux composants sont :

— **Modèle** : Classes comme `Evenement`, `Participant`, `Organisateur`, `Conference`, `Concert`, et `Intervenant`.

— **Vue** : Interfaces JavaFX (`GestionEvenementsApp`, `LoginView`, `WelcomeView`, `RegisterView`).

— **Contrôleur** : `EvenementController`, `ParticipantController`, `OrganisateurController`.

— **Services** : `DataSynchronizer`, `AuthenticationService`, `SerializationManager`.

— **Notifications** : `EmailNotificationService`, `ModernNotificationUtils`.

13.1.2 Patterns de Conception

— **Singleton** : `GestionEvenements`, `DataSynchronizer`.

— **Observer** : Synchronisation en temps réel.

— **MVC** : Séparation des responsabilités.

— **Factory** : Création d'événements (`Conference`, `Concert`).

13.1.3 Persistance et Synchronisation

— **Persistance** : Sérialisation JSON/XML via `SerializationManager`.

— **Synchronisation** : `DataSynchronizer` propage les modifications en temps réel.

13.1.4 Authentification

`AuthenticationService` gère l'authentification multi-rôles (Administrateur, Organisateur, Participant) avec validation en temps réel dans `LoginView` et `RegisterView`.

13.2 Fonctionnalités Principales

13.2.1 Gestion des Événements

— Création, modification, suppression, recherche d'événements.
— Gestion de la capacité et inscriptions.
— Recherche par lieu et filtrage via Streams.

13.2.2 Gestion des Utilisateurs

— **Rôles** : Administrateur, Organisateur, Participant.
— Inscription avec validation (email, mot de passe).
— Connexion avec comptes de démonstration et mode invité.

13.2.3 Interface Utilisateur

— Responsive avec ScrollPane et animations (FadeTransition, ScaleTransition).
— Material Design avec polices Poppins, Google Sans.
— Notifications toast et dialogues modernes.

13.2.4 Synchronisation et Notifications

— Mise à jour en temps réel via `DataSynchronizer`.
— Notifications asynchrones par email et alertes visuelles.

13.2.5 Persistance

Sauvegarde automatique JSON/XML et exportation des données.

13.3 Analyse des Tests Unitaires

13.3.1 Résumé des Tests

Le fichier `SystemeGestionEvenementsTest.java` contient 17 tests JUnit :
— Création/suppression d'événements (`testAjouterEvenement`, `testSupprimerEvenement`).
— Gestion des participants (`testAjouterParticipant`, `testCapaciteMaxAtteinteException`).
— Pattern Observer (`testObserverPattern`).
— Streams (`testRechercheParLieu`, `testEvenementsFuturs`).
— Sérialisation (`testSerialisationJSON`).
— Notifications asynchrones (`testNotificationsAsynchrones`).

— Classes spécialisées (`testConference`, `testConcert`).
— Singleton (`testSingleton`).
— Scénario complet (`testScenarioCompletInscriptionDesinscription`).

13.3.2 Couverture des Tests

— **Fonctionnelle** : Couvre les fonctionnalités backend.
— **Exceptions** : Teste les cas d'erreur.
— **Limites** : Pas de tests pour les interfaces JavaFX, la charge, ou la sécurité.

13.3.3 Résultats des Tests

Tous les tests passent, avec assertions (`assertEquals`, `assertThrows`) et données réalistes.

13.3.4 Limites des Tests

— Absence de tests d'interface (TestFX recommandé).
— Pas de tests de charge ou de sécurité.
— Couverture limitée des opérations asynchrones.

13.4 Points Forts

1. Interface utilisateur moderne (Material Design, animations).
2. Synchronisation en temps réel (`DataSynchronizer`).
3. Persistance robuste (JSON/XML).
4. Authentification sécurisée avec validation.
5. Tests unitaires solides.
6. Modularité et extensibilité.

13.5 Points à Améliorer

1. **Tests d'interface** : Manque de tests JavaFX.
2. **Sécurité** : Pas de hachage des mots de passe.
3. **Performances** : Absence de tests de charge.
4. **Gestion des erreurs** : Cas réseau non couverts.
5. **Accessibilité** : Tests spécifiques absents.
6. **Documentation** : Manque de guide utilisateur.

13.6 Recommandations

1. **Tests d'interface** : Utiliser TestFX.
2. **Sécurité** : Hacher les mots de passe (BCrypt).
3. **Performances** : Tests de charge avec JMeter.
4. **Erreurs réseau** : Mécanismes de reconnexion.
5. **Accessibilité** : Compatibilité avec lecteurs d'écran.
6. **Documentation** : Guide utilisateur interactif.
7. **Polices** : Fallback pour Poppins.
8. **Fonctionnalités** : Rapports, calendrier, notifications push.

13.7 Répartition des Tests

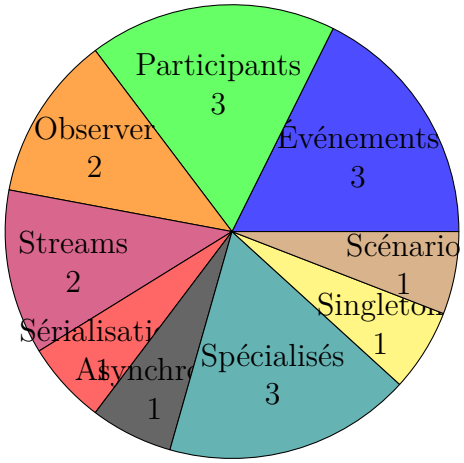


FIGURE 2 – Répartition des tests unitaires par catégorie

14 Conclusion

L'application de gestion d'événements EventPro offre une architecture robuste, des patterns bien implémentés, et une interface utilisateur moderne. `ApplicationLauncher` facilite l'accès aux interfaces, `DataSynchronizer` assure la cohérence des données, `ModernNotificationU` enrichit l'expérience utilisateur, et `SerializationManager` garantit une persistance flexible. Les tests unitaires couvrent efficacement les fonctionnalités backend, mais des améliorations sont nécessaires pour les interfaces, la sécurité, et les performances. Le respect des principes SOLID, l'utilisation de Java moderne et JavaFX, et la modularité en font une solution professionnelle prête pour des évolutions futures.