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 21**, 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.

CONCEPTION (Design & Architecture)

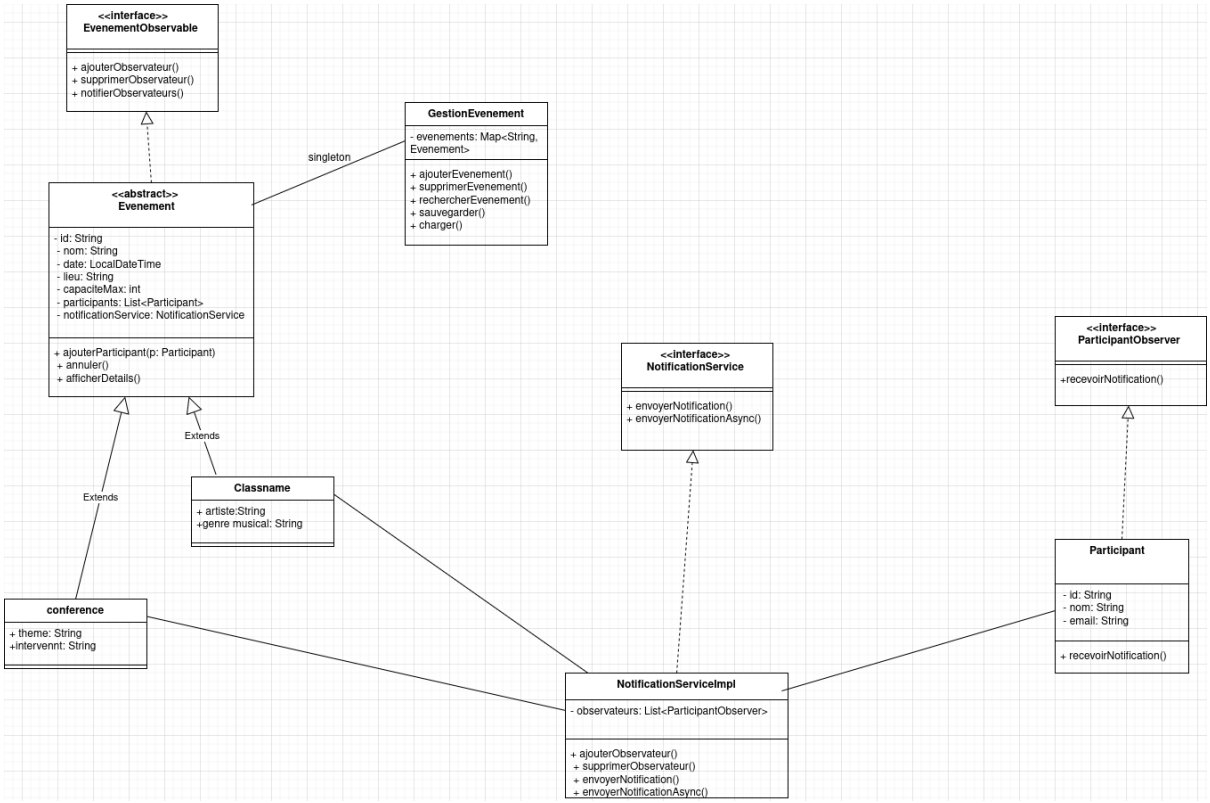
La conception du système repose sur les principes fondamentaux de la **programmation orientée objet (POO)** et l'application de **design patterns** éprouvés. Cette approche garantit une architecture claire, extensible et maintenable.

Diagramme UML

La conception repose sur une modélisation UML mettant en évidence les principales entités du système et leurs relations.

Le diagramme UML ci-dessous illustre les relations entre les principales classes du système :

- Evenement (abstraite), avec ses sous-classes Conference et Concert,
- Participant et Organisateur,
- GestionEvenements (singleton),
- Les interfaces NotificationService, EvenementObservable, ParticipantObserver.



Explication des choix architecturaux

Héritage et polymorphisme

La classe abstraite Evenement encapsule les attributs et comportements communs aux événements. Elle est étendue par Conference et Concert, permettant de gérer différents types d'événements tout en maintenant une interface commune. Cela favorise la réutilisation de code et la cohérence du système.

Pattern Observer

Le système utilise le pattern **Observer** pour notifier automatiquement les participants lorsqu'un événement est annulé ou modifié. Les Participant implémentent l'interface ParticipantObserver et reçoivent les notifications générées par les événements, assurant une réactivité en temps réel.

Singleton (GestionEvenements)

GestionEvenements est conçu comme un **singleton** pour centraliser la gestion des événements. Cela évite la duplication des données et garantit l'accès cohérent au système depuis n'importe quelle partie du programme.

Sérialisation JSON avec Jackson

La persistance des données est assurée grâce à la **sérialisation et désérialisation JSON** en utilisant la bibliothèque **Jackson**. Cela permet de sauvegarder et de recharger l'état du système de manière simple et standardisée, sans recourir à une base de données.

IMPLEMENTATION

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.

Le pattern **Singleton** a été appliqué à la classe GestionEvenements, permettant de centraliser la gestion des événements tout en assurant qu'une seule instance soit utilisée dans le système. Ce singleton propose des méthodes pour ajouter, supprimer ou rechercher des événements, tout en gérant les exceptions spécifiques comme EvenementDejaExistantException.

Le pattern **Observer** a été implémenté via l'interface ParticipantObserver et sa concrétisation dans la classe Participant. Lorsqu'un événement est annulé ou modifié, la méthode notifierObservateurs déclenche l'envoi d'une notification à tous les participants concernés. Pour rendre ce mécanisme encore plus réaliste, une version asynchrone a été intégrée grâce à CompletableFuture, simulant l'envoi différé des notifications.

La persistance des données a été assurée par la sérialisation et la désérialisation en format JSON, en utilisant la bibliothèque **Jackson**. La classe EvenementSerializer prend en charge la sauvegarde et le chargement des événements, tout en respectant les contraintes de typage héritées du modèle objet.

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.

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.
- Mise en place d'une **interface graphique conviviale** (ex : JavaFX) pour faciliter l'utilisation et l'interaction avec le système.

- 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.

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.