

Projet final POO2

Système de gestion d’évènement

26 mai 2025

Mvesso ANDRE

24p757

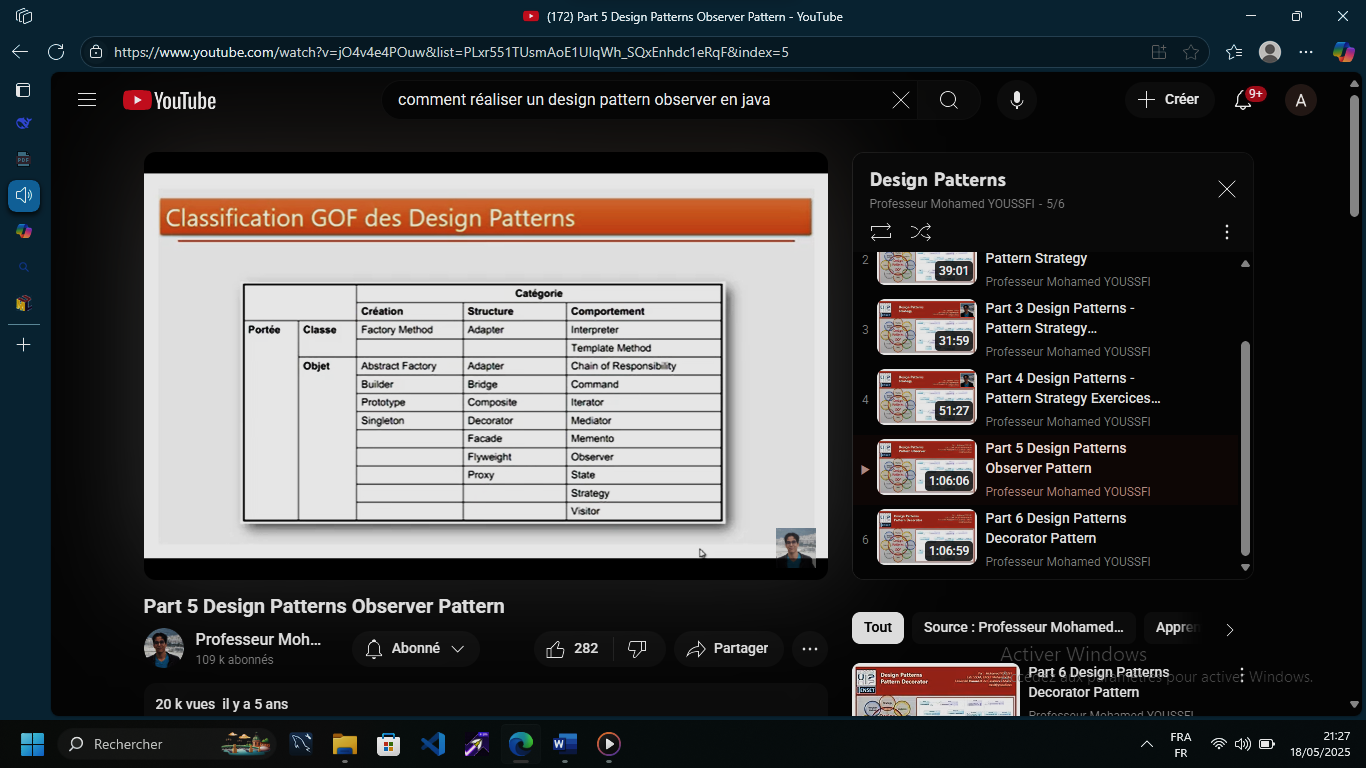
**Introduction**

Dans le cadre de ce projet, nous nous proposons de concevoir un système de gestion d'événements distribué, intégrant des concepts avancés de la Programmation Orientée Objet (POO). Ce projet vise dans un premier temps à structurer et organiser efficacement des événements tels que des conférences, concerts et autres rassemblements avec une gestion optimisée des participants, organisateurs et notifications. Dans un second temps à fortifier nos bases reçus lors du cours de POO, en mettant en œuvre des design patterns fondamentaux tels que Observer, Singleton et pour garantir une architecture modulaire et évolutive. De plus, nous exploiterons la sérialisation JSON/XML afin d'assurer la persistance des données, et nous intégrerons la programmation événementielle et asynchrone pour offrir une expérience réactive et dynamique aux utilisateurs. En n’oubliant pas les notions de bases tels que l’héritage, la création et manipulation d’interfaces, de classe abstraite.

**Design Pattern**

Les Design Patterns (DP) sont des architectures de classes permettant d'apporter une solution à des problèmes fréquemment rencontrés lors des phases d'analyse et de conception d'applications.

Ces solutions sont facilement adaptables (donc réutilisables), elles sont utilisables sans aucun risque dans la grande majorité des langages de programmation orientés objet.



Dans notre cas nous allons utiliser le Design Pattern Observer. Pourquoi ?

Permet de définir une dépendance entre plusieurs objets, telle que, lorsqu'un des objets (le sujet) change d'état (modification de certains de ses attributs), les autres objets (observers) en soit avertis immédiatement.

**Utilisation** : quand un objet doit faire de la notification à d'autres sans faire d'hypothèse sur la nature de ces autres.

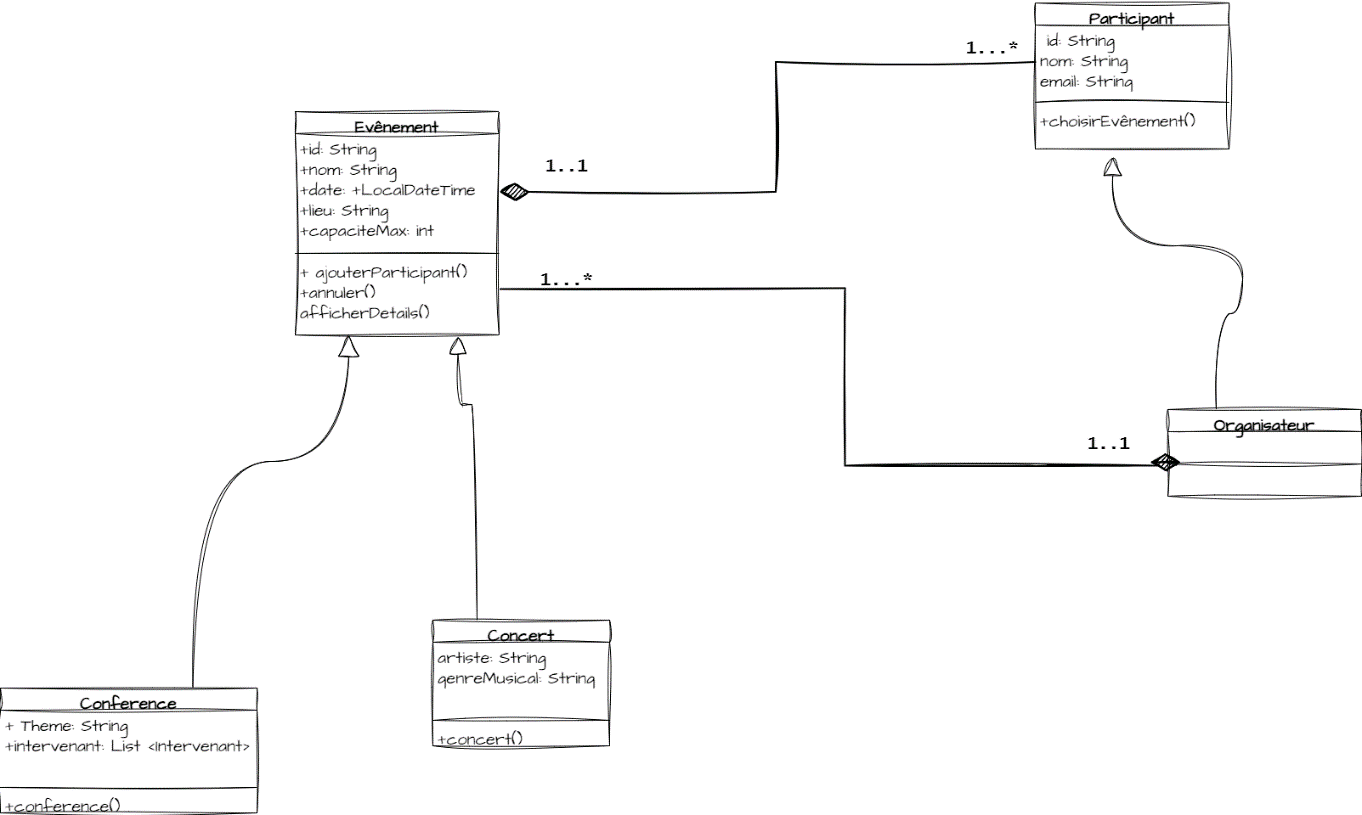
**Exemple** : des données qui sont visualisées dans un tableur et sous forme graphique : diagramme, histogramme... Quand les données sont modifiées toutes les représentations doivent en être informées, sans qu'on construise de dépendance entre ces classes.

**Dans notre cas**

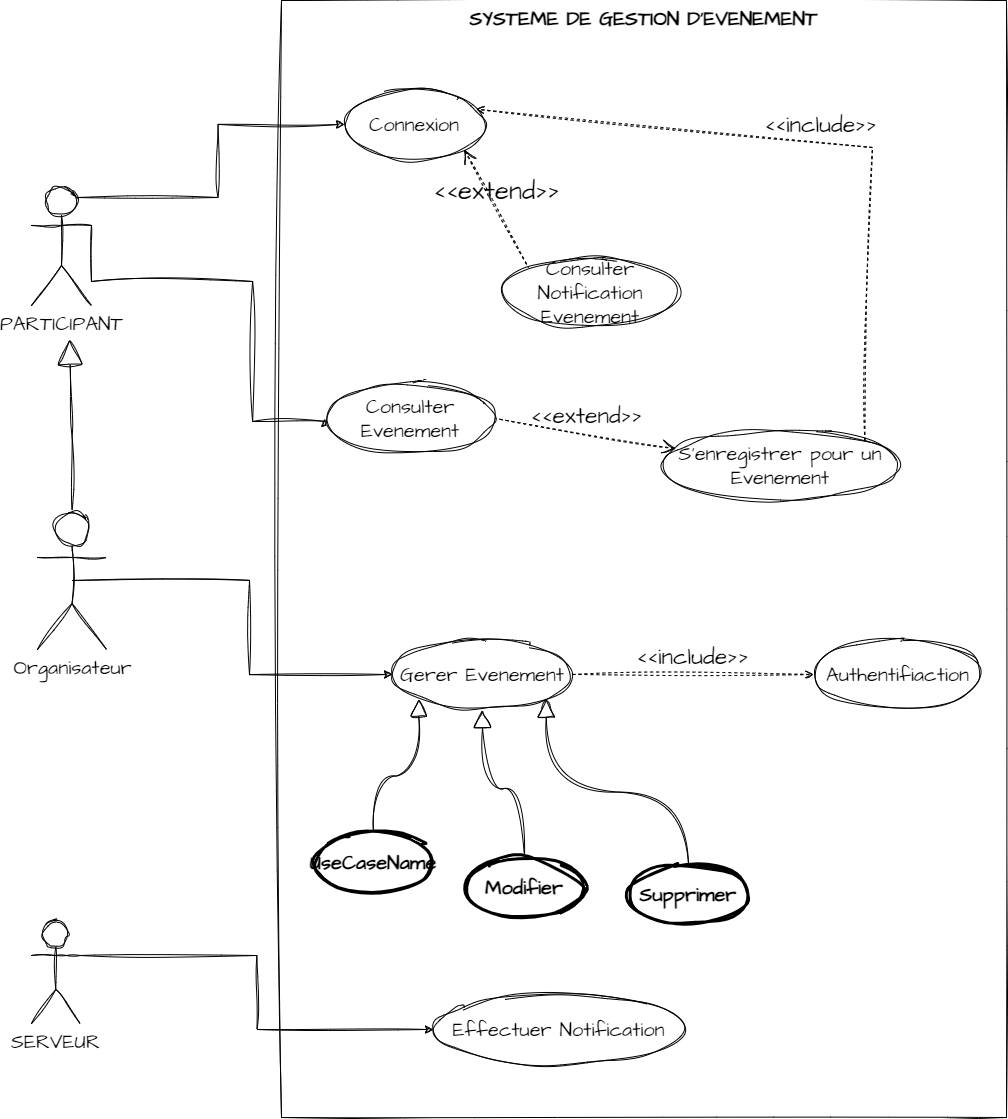
* L’observable est L’Evènement ; pourquoi ? C'est l'objet qui émet des notifications lorsqu'un changement survient (ex : annulation, modification).

Il maintient une liste d'observateurs à notifier.

* L’observateur est le Participant (& Organisateur) C'est l'objet qui reçoit les notifications et réagit (ex : afficher un message). Il "écoute" les changements de l'Evènement et exécute une action



*Diagramme de classes metiers*



*Diagramme de cas d’utilisation*

**Sérialisation**

La **sérialisation** est un mécanisme essentiel en programmation, permettant de convertir un objet en une **représentation textuelle ou binaire**, afin de le **stocker**, le **transmettre** via un réseau ou le **sauvegarder dans un fichier**. Elle est souvent utilisée pour **préserver l’état d’un objet**, facilitant ainsi la persistance et la communication entre systèmes.

En **Java**, la sérialisation peut être réalisée de différentes manières, notamment avec les formats **JSON et XML**, qui sont largement utilisés dans le développement logiciel.

* **Stocker des données de manière structurée** (base de données, fichier local).
* **Envoyer des objets via une API ou un réseau** (communication client-serveur).\*
* **Sauvegarder et recharger l’état d’un programme** (persistance des objets).

Elle est particulièrement utile dans les systèmes distribués, les applications web et les solutions nécessitant une **interopérabilité entre différentes plateformes**.

**Comparaison JSON vs XML pour la sérialisation**

Ces deux formats permettent de stocker des objets, mais présentent des différences clés :

| **Critère** | **JSON** | **XML** |
| --- | --- | --- |
| **Lisibilité** | Plus simple et direct | Plus verbeux |
| **Taille du fichier** | Compact | Plus lourd |
| **Performance** | Rapide | Plus lent |
| **Utilisation principale** | APIs Web, stockage léger | Transactions complexes, structuration hiérarchique |
| **Interopérabilité** | Facile avec JavaScript et Web | Supporte des structures avancées |
| **Gestion des types** | Pas de typage strict | Définit clairement les types de données |

**JSON est souvent préféré** pour sa **simplicité et son efficacité**, tandis que **XML reste puissant** pour des données **structurées** et des systèmes **interopérables**.

**Conclusion de taches réalisées**

Partie 1 : Modélisation de quelques class Classes

Quelques classes

✅1. Evenement (abstraite)

- id: String

- nom: String

- date: LocalDateTime

- lieu: String

- capaciteMax: int

- Méthodes : ajouterParticipant(), annuler(), afficherDetails()

✅2. Conference (hérite de Evenement)

- theme: String

- intervenants: List<Intervenant>

✅3. Concert (hérite de Evenement)

- artiste: String

- genreMusical: String

✅4. Participant

- id: String

- nom: String

- email: String

✅5. Organisateur (hérite de Participant)

- evenementsOrganises: List<Evenement>

✅6. NotificationService (Interface)

- Méthodes : envoyerNotification(String message)

✅7. GestionEvenements (Singleton)

- evenements: Map<String, Evenement>

- Méthodes : ajouterEvenement(), supprimerEvenement(),

- rechercherEvenement()Partie 1 : Modélisation des Classes (UML)

Partie 2 : élément d’Implémentation

✅1. Utilisation du Design Pattern Observer (Faites des recherches à ce sujet)

- Créer un système de notification où les participants sont notifiés lorsqu'un

événement est annulé ou modifié.

- Implémenter EvenementObservable et ParticipantObserver.

✅2. Gestion des Exceptions Personnalisées

- Créer des exceptions comme CapaciteMaxAtteinteException ou

EvenementDejaExistantException.

✅3. Sérialisation JSON/XML

- Sauvegarder et charger la liste des événements dans un fichier JSON/XML en

utilisant Jackson ou JAXB.

4. Utilisation de Streams et Lambdas

5. Bonus : Programmation Asynchrone

- Simuler l'envoi de notifications en différé avec CompletableFuture

Partie 3 : Tests et Validation

Écrire des tests unitaires avec JUnit pour vérifier :

✅- L'inscription/désinscription d'un participant.

✅- La gestion des exceptions.

- La sérialisation/désérialisation.