Rapport de Travail Pratique : Système de Gestion d'Événements

TSAGUE TONFACK Naomi Lucresse

26 mai 2025

Table des matières

1 Introduction

Le présent travail pratique a pour objectif de concevoir et d'implémenter un système distribué de gestion d'événements, englobant notamment des conférences et des concerts. Ce projet s'inscrit dans la volonté d'appliquer et de maîtriser les concepts avancés de la programmation orientée objet (POO), fondamentaux dans le développement logiciel moderne. L'objectif principal est de développer une application capable de gérer efficacement les inscriptions des participants, la gestion des organisateurs, ainsi que la diffusion de notifications en temps réel. Ce système met en œuvre plusieurs principes essentiels de la POO tels que l'héritage, le polymorphisme et l'utilisation d'interfaces, afin de garantir modularité et extensibilité. De plus, le projet intègre des design patterns reconnus (Observer, Singleton, Factory, Strategy) qui favorisent une architecture propre, maintenable et évolutive. Il comprend également la gestion d'exceptions personnalisées afin d'améliorer la robustesse et la fiabilité de l'application. Par ailleurs, la manipulation de collections génériques, la sérialisation et désérialisation des données au format JSON sont exploitées pour assurer la persistance et la portabilité des informations. Enfin, une approche de programmation asynchrone est adoptée pour simuler l'envoi différé des notifications, améliorant ainsi la réactivité et l'expérience utilisateur.

2 Analyse et conception

2.1 Modélisation des classes principales

La conception du système repose sur une modélisation orientée objet structurée et claire, qui vise à représenter de manière fidèle et extensible les entités essentielles liées à la gestion d'événements.

Classe abstraite `Evenement`

Au cœur du modèle se trouve la classe abstraite `Evenement`, qui définit le concept générique d'un événement. Cette abstraction permet d'encapsuler les attributs et comportements communs à tous types d'événements, tout en imposant aux classes dérivées de spécifier leurs particularités.

- **Attributs communs :**
  - `id (String)` : un identifiant unique permettant d'identifier chaque événement.
  - `nom (String)` : le nom de l'événement.
  - `date (LocalDateTime)` : la date et l'heure à laquelle l'événement a lieu.
  - `lieu (String)` : le lieu physique ou virtuel de l'événement.
  - `capaciteMax (int)` : la capacité maximale de participants autorisés.
  - `participants (List<Participant>)` : la liste des participants inscrits à l'événement.
  - `annule (boolean)` : un indicateur pour signaler si l'événement a été annulé.

— **Méthodes principales :**
  — `ajouterParticipant(Participant p)` : permet d'inscrire un participant, avec gestion des capacités.
  — `annuler()` : méthode pour annuler un événement, déclenchant des notifications.
  — `afficherDetails()` (méthode abstraite) : impose aux sous-classes de fournir une description détaillée propre à leur type d'événement.

Cette abstraction facilite la gestion homogène des événements dans la couche applicative et permet une extension future pour d'autres types d'événements sans modifier la structure existante.

Sous-classes concrètes

Deux classes concrètes héritent de `Evenement` pour représenter des types spécifiques d'événements :

— **Conference** Cette classe modélise une conférence, avec des attributs supplémentaires pour décrire ses particularités :
  — `theme (String)` : le thème principal abordé durant la conférence.
  — `intervenants (List<Intervenant>)` : une liste des intervenants qui animent la conférence.

  La méthode `afficherDetails()` est surchargée pour afficher, en plus des informations communes, le thème et les intervenants, offrant ainsi une description complète et contextualisée.

— **Concert** Cette classe représente un concert, avec des attributs spécifiques tels que :
  — `artiste (String)` : le nom de l'artiste ou du groupe se produisant.
  — `genreMusical (String)` : le genre musical joué durant le concert.

  De même, la méthode `afficherDetails()` fournit une description détaillée intégrant ces informations propres au concert.

Classe `Participant`

La classe `Participant` modélise les utilisateurs qui s'inscrivent aux événements. Chaque participant possède :

— `id (String)` : identifiant unique.

— `nom (String)` : nom du participant.

— `email (String)` : adresse email utilisée pour la communication.

Cette classe implémente l'interface `ParticipantObserver`, ce qui lui permet de recevoir des notifications en temps réel, notamment en cas d'annulation ou de modification d'événement auxquels il est inscrit. Cette approche utilise le design pattern Observer pour assurer une communication fluide et découplée.

Classe `Organisateur`

L'organisateur est une spécialisation de participant, héritant donc des propriétés de base de la classe `Participant`. Il possède en outre :

— `evenementsOrganises (List<Evenement>)` : une collection des événements qu'il administre.

Cette distinction permet de gérer des rôles différents dans le système, où l'organisateur a des droits étendus, notamment pour créer, modifier ou annuler des événements. Cela facilite également la traçabilité et la gestion des événements depuis leur origine.

2.2 Gestion des événements

La gestion centralisée des événements est assurée par la classe `GestionEvenements`, conçue selon le design pattern **Singleton**. Ce choix architectural garantit l'existence d'une seule instance unique de gestionnaire tout au long de l'exécution du programme, ce qui évite les incohérences liées à une gestion décentralisée ou à la multiplication des gestionnaires.

Rôle principal

`GestionEvenements` joue le rôle de contrôleur central du système, chargé de maintenir et de manipuler la collection complète des événements existants. Cette centralisation facilite l'organisation, la recherche, la modification et la suppression des événements de manière cohérente et sécurisée.

Implémentation du Singleton

— La classe encapsule un attribut statique privé contenant l'instance unique.

— Le constructeur est privé pour empêcher toute instanciation extérieure.

— Une méthode publique statique `getInstance()` fournit un accès global à l'unique instance, créant cette dernière au besoin (instanciation paresseuse).

Cette approche assure une gestion centralisée, accessible partout dans le système, sans risque de créer plusieurs gestionnaires indépendants pouvant causer des divergences dans les données.

Gestion de la collection d'événements

Les événements sont stockés dans une structure de données de type `Map<String, Evenement>`, où la clé est l'identifiant unique de l'événement (`id`), permettant un accès rapide et efficace.

Les méthodes principales exposées sont :

— `ajouterEvenement(Evenement e)` Permet d'ajouter un nouvel événement à la collection. Cette opération vérifie si l'identifiant existe déjà pour éviter les doublons, et déclenche une exception personnalisée (`EvenementDejaExistantException`) en cas de conflit.

— `supprimerEvenement(String id)` Supprime l'événement identifié, avec gestion d'erreur si l'événement n'existe pas.

— `rechercherEvenement(String id)` Recherche un événement par son identifiant.

— `getListeEvenements()` Retourne la liste complète des événements.

2.3 Inscription et notifications

Inscription des participants

L'inscription à un événement s'effectue via la méthode `ajouterParticipant(Participant p)` dans la classe `Evenement`. Cette méthode contrôle la capacité maximale et empêche les inscriptions au-delà. En cas de dépassement, une exception `CapaciteMaxAtteinteExceptio` est levée.

Les participants sont alors automatiquement enregistrés dans la liste de participants de l'événement.

Notifications et design pattern Observer

Pour assurer la communication dynamique entre les événements et les participants, le pattern **Observer** est mis en œuvre :

— `Evenement` agit comme `Subject` : il garde la liste des observateurs (participants) et les notifie des changements importants.

— `Participant` agit comme `Observer`, recevant des notifications notamment en cas d'annulation ou modification de l'événement.

Lorsqu'un événement est annulé via la méthode `annuler()`, tous les participants inscrits sont informés par l'appel à leur méthode `update()`, leur permettant ainsi de recevoir une notification en temps réel.

2.4 Persistance des données

La sauvegarde et le chargement des données sont réalisés à l'aide de la sérialisation JSON, assurant une portabilité et une interopérabilité améliorées.

— Les événements et leurs participants sont convertis en JSON via des bibliothèques telles que Gson ou Jackson.

— Le fichier `evenements.json` stocke l'ensemble des événements.

— Le chargement du fichier au démarrage permet de restaurer l'état précédent, et la sauvegarde périodique ou déclenchée à des moments clés garantit la pérennité des données.

2.5 Gestion des exceptions personnalisées

Pour renforcer la robustesse, plusieurs exceptions spécifiques sont définies, par exemple :

— `EvenementDejaExistantException` : levée lors de l'ajout d'un événement avec un identifiant déjà utilisé.

— `CapaciteMaxAtteinteException` : levée lorsque la capacité maximale d'un événement est atteinte.

— `EvenementIntrouvableException` : lorsqu'une opération sur un événement inexistant est demandée.

Cette gestion explicite des erreurs améliore la lisibilité, facilite la détection des anomalies, et rend le système plus stable.

2.6 Programmation asynchrone et notifications différées

Afin de simuler l'envoi différé des notifications, une stratégie asynchrone a été intégrée. L'interface `NotificationStrategy` définit la méthode d'envoi, avec au moins deux implémentations possibles :

— `NotificationImmediate` : envoi instantané des messages.

— `NotificationRetardee` : envoi différé simulé, avec un délai arbitraire.

Ce mécanisme améliore la modularité et ouvre la voie à des extensions futures, comme l'intégration de vraies notifications réseau ou via SMS.

2.7 Tests unitaires

Le projet comprend une suite de tests unitaires basés sur JUnit, permettant de valider le bon fonctionnement des principales fonctionnalités :

— Ajout, suppression et recherche d'événements.

— Inscription des participants, avec gestion des capacités.

— Annulation d'événements et réception des notifications.

— Gestion des exceptions personnalisées.

— Sérialisation et désérialisation JSON.

Ces tests assurent une base solide pour garantir la qualité du code.

3 Conclusion

Ce travail pratique a permis d'appliquer les concepts avancés de la programmation orientée objet dans un contexte réaliste, en développant un système distribué complet de gestion d'événements. L'architecture retenue, fortement modulaire et basée sur des design patterns éprouvés, garantit la maintenabilité et l'évolutivité du projet. Les fonctionnalités mises en place, notamment l'inscription des participants, la gestion des organisateurs, la persistance JSON, et les notifications en temps réel, répondent aux exigences initiales et offrent un socle solide pour d'éventuelles extensions ultérieures.