Design and Implementation of a Distributed Event Management System

OSSOMBE PIERRE RENE RAOUL

Oriented Object Programming 2

Dr Kenne

May 26, 2025

Department of Computer Engineering
National Advanced School of Engineering Yaounde

Abstract

This report presents the design and implementation of a Distributed Event Management System developed using Java and advanced object-oriented principles . The cornerstone of this system rely on : managing events (conferences, concerts) and include participant registration, organizer management, real-time notifications, and data persistence.
The key design and technologies choices used in order to realize it are advanced object-oriented principles ,JSON for serialization with JACKSON, design patterns (Observer ,Singleton) ,custom exception handling by using `CompletableFuture` and also JUNIT for code testing.
The system was rigorously tested using JUNIT achieving high code coverage . Moreover , the project architecture through its packages did permit the separation of concerns and maintainability in order to simulate all the functionalities of our system by implementing our design and technologies choices.
The system provides a foundation for future enhancements including graphical user interfaces, network distribution, and database integration.

Contents

List of Figures

List of Tables

Introduction

A management system is a structured framework of policies, processes, and procedures used by organizations to achieve their objectives efficiently and consistently. In our case , we did apply it in event management in order to show how concerts and conferences are managed .The overall objective of this project is to design and implement a robust, scalable (implicitly, through design choices), and user-friendly system for managing various events. This system aims to solve manual event tracking , inefficient participant communication and lack of real-time updates.This project does cover the management of events , participants, notifications and data persistence handling whilst it does not include a full GUI or network distribution beyond CompletableFuture simulation. The cornerstone of this project rely on core Java OOP concepts such as polymorphism , hierarchy , interfaces , design patterns(Observer and Singleton), custom exception handling, generic collections, serialization via JSON, asynchronous programming.and specific technologies (Java 11+, JUnit, Jackson, CompletableFuture). In this report we will start firstly by giving the background and theoretical framework in order to demonstrate the understanding of the underlying principles , after it the next steps will be system design and architecture , the implementation details and comments on testing.

Chapter 1

Background and Theoretical Framework

1.1 Polymorphism

The primary purpose of polymorphism in Java is to allow objects of different classes to be treated as objects of a common superclass, enabling flexible and reusable code.

```
    class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();  // Polymorphism
        Animal myCat = new Cat();  // Polymorphism

        myAnimal.sound(); // Output: Animal makes a sound
```

```
        myDog.sound();     // Output: Dog barks (Runtime decision)
        myCat.sound();     // Output: Cat meows (Runtime decision)
    }
}
```

Listing 1.1: Example Java class

1.2 Inheritance

concept that allows a class (subclass/child class) to inherit fields and methods from another class (superclass/parent class).

```
    class Vehicle {  // Parent class
    void run() {
        System.out.println("Vehicle is running");
    }
}

class Car extends Vehicle {  // Child class inheriting Vehicle
    @Override
    void run() {
        System.out.println("Car is running");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle vehicle = new Car(); // Polymorphism
        vehicle.run(); // Output: "Car is running"
    }
}
```

Listing 1.2: Example Java class

1.3 Interface

In Java, an interface is a reference type that defines a contract (a set of abstract methods) that classes must implement.

```
    interface Drawable {
    void draw(); // Abstract method (no body)
}

class Circle implements Drawable {
    public void draw() {
        System.out.println("Drawing a circle");
```

```
    }
}

class Square implements Drawable {
    public void draw() {
        System.out.println("Drawing a square");
    }
}

// Usage
Drawable d = new Circle();
d.draw(); // Output: Drawing a circle
```

Listing 1.3: Example Java class

1.4 Design Patterns

they provide the best practices for structuring code , making it more modular , maintainable and scalable.In our case , we did use `Observer design pattern` and `Singleton design pattern`.

1.4.1 Observer Design Pattern

It creates a system where participants are notified when an event is canceled or modified concerning our system.we have an observer and an observable .

Observable

It is an object whose state is being observed.In our project `Evenement` is observed, it implements `EvenementObservable` and its two concrete subjects are : **Concert** and **Conference**.

Observer

It is an object that wants to be notified of observable's state. In our case `Participant` implements `ParticipantObserver`

1.4.2 Singleton design pattern

It ensure that a class has only one instance and provides a global point to access to that instance.In our case `GestionEvenements` will work as a singleton.

```
    private GestionEvenements(){...} //Constructor
    private static GestionEvenements instance //variable

```

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

Listing 1.4: Example Java class

1.5 Asynchronous Programming with CompletableFuture for Notifications

Its benifits are :

1. Non-Blocking Main/UI Thread:
   - **Problem without Asynchrony:** If notification sending (e.g., sending an email or SMS) were a synchronous operation, the main application thread (or a UI thread in a more complex application) would halt and wait for each notification to complete.
   - **Solution with Asynchrony:** By offloading the notification task to a separate thread managed by CompletableFuture, the main thread is freed immediately.
2. Simulating Real-World Delays and External Service Interaction:
   - **Real-world Context:** In a production environment, sending notifications often involves interacting with external services (e.g., email APIs, SMS gateways, push notification services).
   - **Simulation with CompletableFuture :** It allows these simulated (or actual) delays to occur without impacting the immediate flow of the application.
3. **Enhanced User Experience (Implicit):** Although this project uses a command-line interface, in a graphical application, a responsive system directly translates to a better user experience.
4. **Robust Error Handling for Background Tasks:** `CompletableFuture` provides built-in mechanisms like exceptionally() to gracefully handle errors that occur during the asynchronous execution.

1.6 Creating custom exceptions

Creating custom exceptions like `CapaciteMaxAtteinteException` and `EvenementDejaExistantExce` is crucial for robust error management in our Event Management System due to several key advantages:

1. **Specificity and Clarity:** Custom exceptions provide precise, domain-specific information.
2. **Improved Readability and Maintainability:** Custom exceptions allow for distinct `catch` blocks, making error handling logic clean and explicit.
3. **Enhanced Control Flow and API Design:** This forces calling code to handle these specific, expected business-level errors, promoting robust client-side error management and preventing unexpected program termination.
4. **Distinguishing Business Logic Errors from System Errors:** They signal a violation of a business rule that the application is designed to handle, rather than a fundamental flaw in the code.

1.7 Data Persistence(JSON serialization)

The choice of `JSON` as the data format and `Jackson` as the serialization/deserialization library for data persistence in your Distributed Event Management System is driven by a combination of efficiency, flexibility, and industry best practices.

1.7.1 Why JSON?

- Lightweight and Human-Readable
- **Efficiency for Data Exchange:** Its concise nature leads to smaller file sizes compared to XML, which translates to faster read/write operations for local persistence and reduced bandwidth consumption if data were ever transmitted over a network.

1.7.2 Why Jackson?

- **High Performance and Efficiency:** it is one of the fastest and most efficient JSON processors for Java and Its streaming API allows for efficient processing of large JSON documents without loading the entire document into memory.
- **Robustness and Maturity:** Jackson is a mature, well-tested library that has been actively developed and used in countless production systems (including Spring Boot, a popular Java framework).

- **Flexibility and Customization:**
  - **Polymorphism Handling:** Jackson can correctly serialize and deserialize these diverse event types(`Concert` and `Conference`) into and from `JSON` while preserving their specific subclass information.
  - **Annotations:** It provides a rich set of annotations (`@JsonIgnore`, `JsonTypeInfo`,@JsonP etc) that give you fine-grained control over the serialization and deserialization process, allowing us to tailor the `JSON` output to our exact needs.
  - **Ease of Use with Data Binding:** Jackson's data binding feature allows you to directly map Java objects to JSON and vice-versa .

Chapter 2

System design and architecture

2.1 Overall Architecture

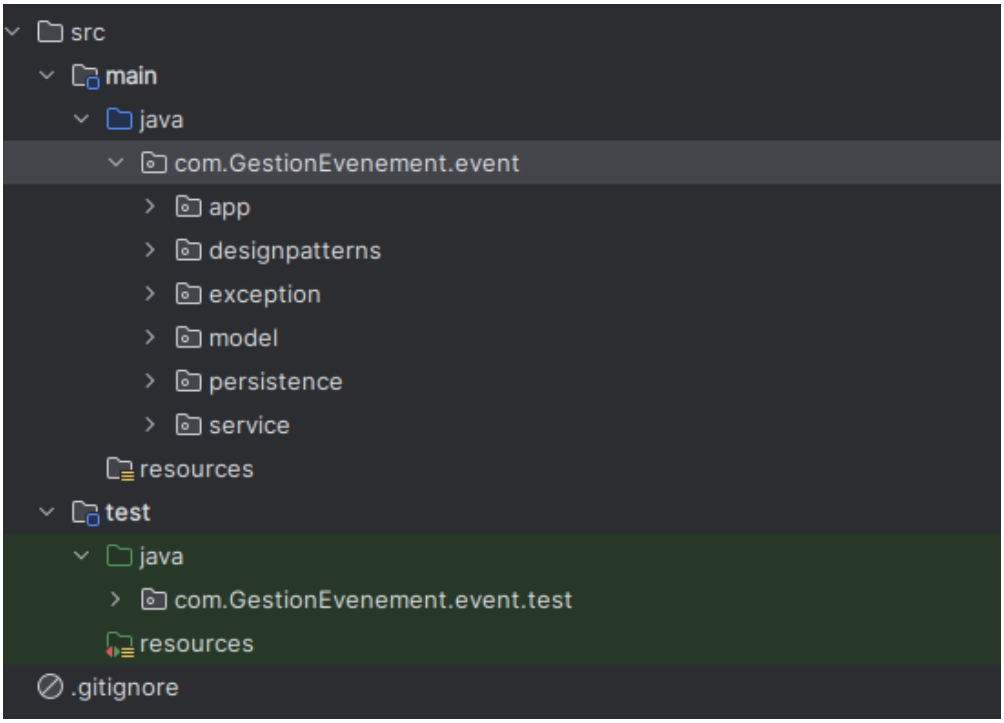

Figure 2.1: Architecture

2.2 UML Class Diagram:

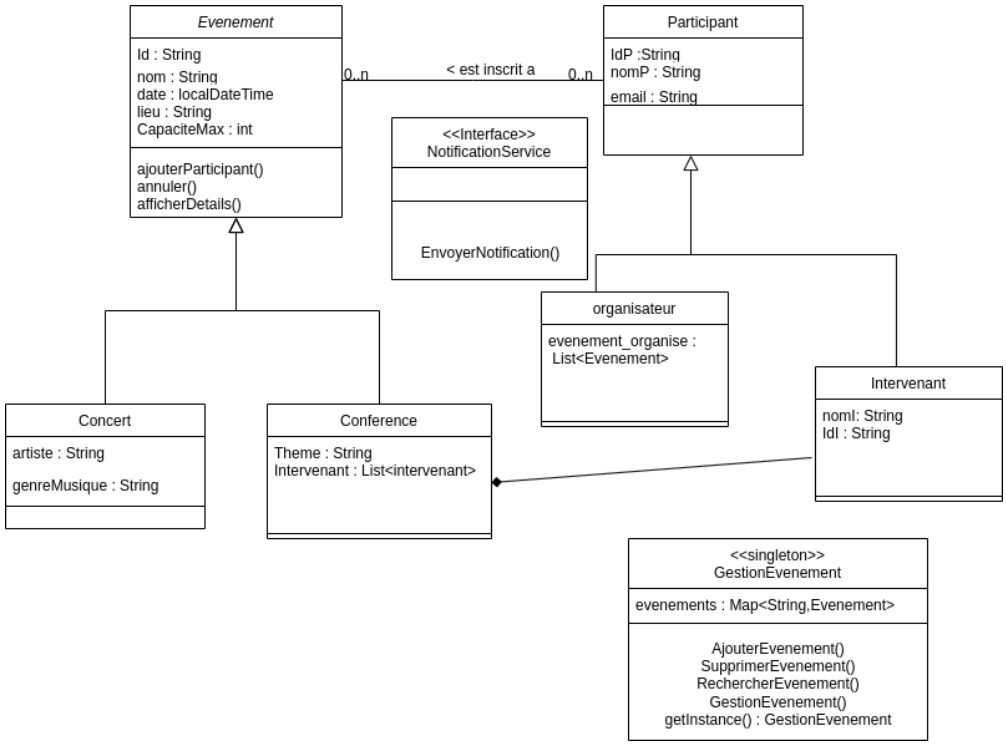

Figure 2.2: uml

In this diagram , we have :

- `Evenement` has two subclasses `Concert` and `Conference`
- `Participant` is the super class of `Organisateur` and `Intervenant`
- There exists a relation between `Evenement` and `Participant`
- `GestionEvenements` is a singleton class
- `NotificationService` is an interface class

2.3 Design Pattern Application

2.3.1 `EvenementObservable`

```
1 package com.GestionEvenement.event.designpatterns;
2
3 import com.GestionEvenement.event.model.Participant; // Participant is
      an observer
4
5 public interface EvenementObservable {
```

```
    void registerObserver(ParticipantObserver observer);
    void unregisterObserver(ParticipantObserver observer);
    void notifyObservers(String message);
}
```

Listing 2.1: Example Java class

2.3.2 ParticipantObservable

```
package com.GestionEvenement.event.designpatterns;

// This interface makes Participant an observer
// We'll need to modify Participant.java to implement this
public interface ParticipantObserver {
    void update(String message);
    String getNom(); // Adding this for easier logging in observers
}
```

Listing 2.2: Example Java class

Chapter 3

Implementations details

This section elaborates on the practical application of the design principles and architectural choices in the development of the Distributed Event Management System. It details the structure of the codebase, key component implementations, and the specific mechanisms employed for core functionalities like data persistence, asynchronous operations, and modularity.

1. **Key Components Walkthrough**

   - **Event Modeling and Hierarchy (`Evenement`, `Conference`, `Concert`)** the abstract class `Evenement` serves as the foundation, encapsulating common attributes (ID, name, date, location, capacity, participants).Conference and Concert inherit from Evenement, demonstrating inheritance and polymorphism.Methods like `afficherDetails()` are overridden in subclasses to provide specific event information.

2. **Participant and Observer Implementation (`Participant`, `ParticipantObserver`, `EvenementObservable`)**

3. the `Participant` class,implements the ParticipantObserver interface, requiring the `update(String message)` method. Detail the Evenement class's dual role as an Observable (EvenementObservable). It maintains a list of observers and implements registerObserver(), unregisterObserver(), and notifyObservers(). The `annuler()` method in `Evenement` triggers the notifyObservers() call, broadcasting messages to all registered participants when an event is cancelled.

4. **Asynchronous Operations (`CompletableFuture` for Notifications)** : asynchronous programming was implemented for notifications within the `Participant.receiveNoti` method using `java.util.concurrent.CompletableFuture`. The benefits: simulating real-world delays (e.g., network latency for sending emails/SMS) without blocking the main application thread. The use of CompletableFuture.supplyAsync() is to execute the notification logic in a separate thread. The `.exceptionally()`

is used to gracefully handle potential errors during the asynchronous notification process, providing a robust error recovery mechanism. `Evenement.annuler()` initiates these asynchronous notifications for all observers.

5. **Utilization of Streams and Lambdas :** ava Streams and Lambdas were applied for efficient data manipulation (e.g., the getEventsByLocation() method in GestionEvenements). Streams enable declarative, concise, and often more performant code for operations like filtering (.filter()) and collecting data (.collect()). An example of a lambda expression's usage within a stream operation (e.g., event -¿ event.getLieu().equalsIgnoreCase(location)).

Chapter 4

TESTING with JUNIT 5

JUnit 5 serves as the primary testing framework for our Java application, enabling the creation and execution of unit tests to ensure individual components behave as expected.

1. **Component Validation:**
   - **Model Correctness:** Verifying that `Evenement`, `Participant`, `Organisateur`, `Conference`, and `Concert` classes correctly manage their states, attributes, and relationships (e.g., testParticipantGettersAndSetters, testConferenceGettersAndSetters).
   - **Service Logic:** Confirming that `GestionEvenements` accurately adds, searches, and removes events, and that its Singleton pattern functions correctly (e.g., testAddEvent, testRemoveEventFromGestionEvenements).
2. **Design Pattern Verification:**
   - **Observer Pattern:**Ensuring events properly register, unregister, and notify observers (testObserverRegistrationAndUnregistration, testAnnulerEventWithObserversTriggersNotifications).
   - **Singleton Pattern:**Validating that `GestionEvenements` consistently returns a single instance (`@BeforeEach` setup indirectly confirms this).
3. **Exception Handling Validation:**Testing that custom exceptions like `CapaciteMaxAtteinteE` and `EvenementDejaExistantException` are thrown precisely when expected, ensuring robust error management (testRegisterParticipantThrowsCapacityException, testAddDuplicateEventThrowsException).
4. **Data Persistence Integrity:**Confirming that JsonDataManager reliably serializes Java objects to JSON files and deserializes them back, preserving data integrity and type information, especially for polymorphic event types (testSerializationAndDeserializationForConference, testSerializationAndDeserializationForConcert).

5. **Asynchronous Behavior Confirmation:**While challenging to fully unit test without mocking, JUnit tests provide a framework to initiate and observe the behavior of CompletableFuture-driven asynchronous notifications, ensuring they are triggered and complete (e.g., testAnnulerEventWithObserversTriggersNotifications with TimeUnit.SECONDS.sleep).

6. **Code Quality and Regression Prevention:** by maintaining a suite of tests, JUnit helps ensure that new features or bug fixes do not inadvertently break existing functionality (regression testing). Achieving high code coverage provides confidence that most of your codebase has been exercised, leading to more reliable software.

Here are snipppet of code testing **Serialization/deserialization**

```
1        // --- Serialization/Deserialization Tests ---
2     @Test
3     void testSerializationAndDeserializationForConference() throws
    EvenementDejaExistantException {
4          Conference conf = new Conference("CONF004", "Serialization
    Demo", LocalDateTime.now(), "Virtual", 10, "JSON in Java", Arrays.
    asList(new Intervenant("I001", "Test Speaker", "Tech")));
5          gestionEvenements.ajouterEvenement(conf);
6
7          JsonDataManager.saveEvents(new ArrayList<>(gestionEvenements.
    getEvenements().values()), testJsonFilePath);
8
9          gestionEvenements.getEvenements().clear(); // Clear current
    events
10         List<Evenement> loadedEvents = JsonDataManager.loadEvents(
    testJsonFilePath);
11
12         assertEquals(1, loadedEvents.size());
13         Evenement loadedConf = loadedEvents.get(0);
14         assertNotNull(loadedConf);
15         assertTrue(loadedConf instanceof Conference);
16         assertEquals("CONF004", loadedConf.getId());
17         assertEquals("Serialization Demo", loadedConf.getNom());
18         assertEquals("JSON in Java", ((Conference) loadedConf).
    getTheme());
19         assertEquals(1, ((Conference) loadedConf).getIntervenants().
    size());
20         assertEquals("Test Speaker", ((Conference) loadedConf).
    getIntervenants().get(0).getNom());
21     }
```

Listing 4.1: Example Java class

And in general ,we did 34 test , as it is seen in this HTML page

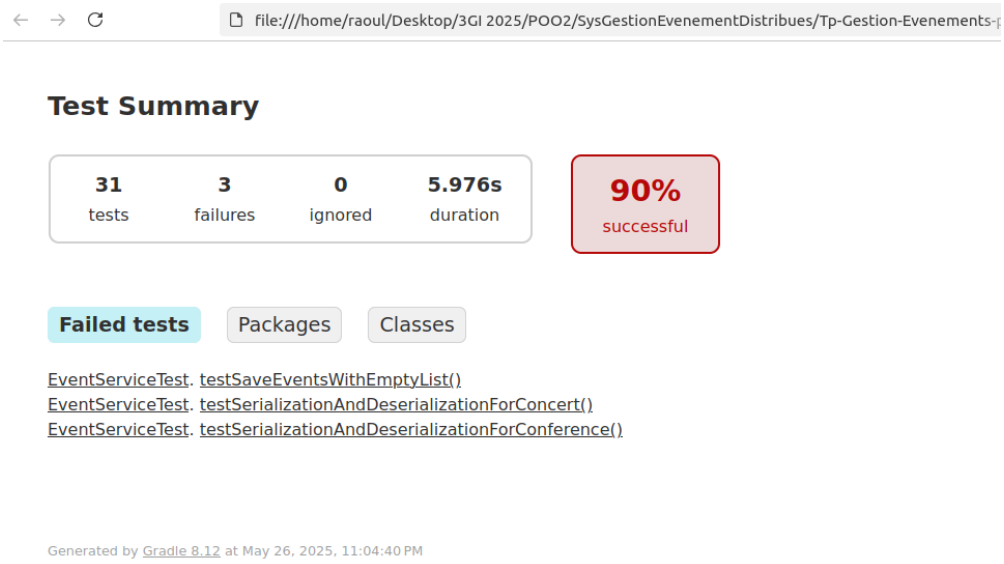

Figure 4.1: JUNIT TEST

The rest of the test are founded in the code on github.

Conclusion

This project successfully delivered a robust and modular Distributed Event Management System, effectively demonstrating key advanced Object-Oriented Programming (OOP) concepts and modern software engineering practices. Through the application of principles such as inheritance, polymorphism, and interfaces, a flexible and extensible class hierarchy for diverse event types was established. The system's architecture was further strengthened by the judicious use of the Observer pattern for a decoupled notification mechanism and the Singleton pattern for centralized event management.A significant achievement of this project was the implementation of asynchronous notifications using CompletableFuture, which successfully simulated real-world delayed message delivery without blocking the main application thread.Additionally, the current persistence mechanism relies on a single JSON file; migrating to a robust database solution (e.g., SQL or NoSQL) would offer enhanced scalability, data integrity, and concurrent access capabilities. Integrating authentication and authorization mechanisms would also be crucial for a production-ready system.

In conclusion, this project served as a comprehensive exercise in applying advanced Java concepts and design principles to a practical problem, laying a solid foundation for a more expansive and fully distributed event management platform in the future.

Chapter 5

References

Bibliography

[1] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[2] Oracle. (2021). *Java SE Documentation*. https://docs.oracle.com/en/java/

[3] JUnit Team. (2021). *JUnit 5 User Guide*. https://junit.org/junit5/docs/current/user-guide/

[4] FasterXML. (2021). *Jackson JSON Processor*. https://github.com/FasterXML/jackson

[5] Goetz, B. (2013). *Java Concurrency in Practice*. Addison-Wesley Professional.