Project Scope – Project Part 2

Student Names: Jeta Sylejmani, Anes Fonda, Adea Krasniqi

Course Name: SW Des Principles and Patterns

Section: 501

Date: 04/12/2025

Project Scope

The EventFlow application is implemented as a Java-based event ticketing system with a layered architecture. It is currently a simple console-based Event Ticketing Platform, where the main focus of this phase is on the back-end logic: domain models, repositories, and services.

A basic console menu is available for interacting with the system, mainly for testing and demonstrating the core use cases. There is no graphical or web user interface yet. Instead, the project emphasizes separation of concerns and the use of basic design patterns (such as Singleton repositories and a service layer for business logic) to keep the system maintainable and easy to extend in future iterations.

In Scope (Implemented in This Phase)

• Core Domain Models (Models layer)

Implementation of Java classes representing the main entities of the system:

- **Event** – stores event information (ID, title, description, location, date).
- **User** – represents attendees with basic account data (ID, name, email, password).
- **Organizer** – represents event organizers with contact details and password.
- **Tickettype** – defines types of tickets per event (name, price, quantity available).
- **Ticket** – represents individual tickets linked to an event and a ticket type, with a sold status.
- **Order** – represents a user's ticket purchase (user, ticket, total price).
- **Payment** – represents a simulated payment for an order (amount, method, success flag, time).
- **Notification** – represents messages sent to users (recipient, message, sent time).

• In-Memory Data Access (Repositories layer)

For each main entity (except Notification, which is handled directly in the service), a corresponding repository is implemented using in-memory List collections and the Singleton pattern:

- EventRepository, UserRepository, OrganizerRepository, TickettypeRepository, TicketRepository, OrderRepository, PaymentRepository.

These repositories provide basic **CRUD-style** operations such as **findAll**, **findById**, findByUserId, findByEventId, save, and deleteById, with auto-incrementing **IDs** managed inside each repository.

• Business Logic (Services layer)

A set of service classes encapsulates the main use cases and rules of the system:

- **EventService**
  - creates events,
  - lists all events,
  - retrieves event details,
  - cancels events.
- **UserService**
  - registers new users,
  - retrieves user information.
- **OrganizerService**
  - creates organizers,
  - lists organizers,
  - finds organizers by ID.
- **TickettypeService**
  - creates ticket types for a given event,
  - retrieves all ticket types,
  - retrieves ticket types by event,
  - finds a ticket type by ID.
- **TicketService**
  - issues tickets,
  - lists all tickets,
  - retrieves tickets by event,
  - validates ticket availability (checks that a ticket exists and is not sold).
- **OrderService**
  - creates orders only if the ticket exists and is not already sold,
  - sets the order's totalPrice based on the ticket price,
  - marks tickets as sold when an order is created,
  - cancels orders and resets the ticket's sold status when possible.
- **PaymentService**
  - processes simulated payments for orders (checks that the amount is at least the order's total price),
  - stores payment records in memory,
  - allows marking a payment as refunded/unsuccessful by setting its success flag to false.
- **NotificationService**
  - sends notifications only if the recipient user exists (validated via UserRepository),
  - assigns IDs to notifications and stores them in an in-memory list.

• Application Entry Point

- Application class with a main method, acting as the starting point for running the program and as a place where a future user interface (CLI or GUI/web) can be integrated with the existing services.

• Design & Architecture Goals

- Clear separation between Model, Repository (data access), and Service (business logic) layers.
- Use of the Singleton pattern for repositories to ensure a single shared data source in memory.
- Code structured so that different kinds of user interfaces can be added later without changing the core business logic in the services and models.

Out of Scope (Future Enhancements – Not Implemented Yet)

These features are not present in the current code but are planned as possible future work:

- **Real Authentication & Security**
  - No password hashing or encryption of sensitive data.
  - No login sessions, tokens, or role-based access control (any menu option can be used by whoever runs the program).
- **Persistence & Data Storage**
  - No database or file storage; all data is stored in in-memory lists and is lost when the application stops.
  - No backups, migrations, or multi-user concurrent access.
- **User Interface**
  - No graphical web UI, mobile app, or desktop GUI.
  - Only a basic console menu is implemented for testing; there is no advanced or user-friendly front-end yet.
- **Real Payment Processing & Integrations**
  - No integration with real payment providers (credit/debit cards, PayPal, bank APIs, etc.).
  - All payments are simulated in memory for demonstration and testing only.
- **Domain Behaviour & Automation**
  - No automatic notifications when orders are created, events are cancelled, or payments are processed (Observer-style event handling is planned but not implemented; notifications are sent only via a manual menu option).
  - Orders currently support only a single ticket per order; there is no shopping cart or group ordering.
  - No separate user-initiated ticket cancellation or refund flow (only event-level cancellation cascades to tickets/payments).
- **Advanced Features**
  - No QR code generation/scanning for tickets.
  - No advanced analytics, dashboards, or detailed reporting.
  - No seat maps, reserved seating, dynamic pricing, or ticket resale/transfer logic.
  - No multi-organization billing, tax handling, or multi-currency support.
- **Robust Validation & Non-Functional Concerns**
  - Input validation is minimal; for example, there is no validation of email format, password strength, or full handling of invalid console input types.

- No logging, auditing, or monitoring.
- No performance, scalability, or concurrency considerations.

Used Patterns

Pattern 1 – Singleton Pattern

Risk (before Singleton):
Without the Singleton pattern, different parts of the system could create their own separate instances of repositories like EventRepository, UserRepository, TicketRepository, etc. That means:

- Each instance would have its own list of data.
- IDs (nextId) could be duplicated.
- Changes made through one repository object would not be visible in another.

This could easily lead to inconsistent data in memory and confusing behavior.

Solution (using Singleton):
By applying the Singleton pattern to each repository (e.g. EventRepository.getInstance()), the system now has only one shared instance per repository.

- All services work with the same data.
- ID generation stays consistent.
- It is easier to keep the state of events, users, tickets, orders, and payments synchronized across the whole application.

Pattern 2 – Observer Pattern (for Notifications – *planned design*)

Note: The current code does not yet implement the Observer pattern, but it is planned as a future improvement for the notification flow.

Risk (before Observer):
Currently, sending notifications depends on code manually calling **NotificationService**. For example, after creating an order or canceling an event, someone has to remember to write extra code like **notificationService.sendNotification(...).** This causes risks such as:

- Some important actions might not send any notification at all.
- Different services might send inconsistent messages for the same type of action.
- Notification logic could become scattered across the code, making it harder to update.

Solution (using Observer – planned):
By introducing an **Observer-style** design, core services (like OrderService, EventService, PaymentService) will be able to publish events (e.g. "orderCreated", "eventCanceled"), and NotificationService will act as an observer that listens for these events.

- Notifications will be sent automatically whenever something important happens.
- Notification logic will be centralized in one place.
- It will be easier to add new reactions (for example, send email, log activity) without changing the core business logic.

Pattern 3 – GRASP Controller (Services as Controllers)

Risk (before GRASP Controller):
Without a dedicated service layer acting as controllers, the main method or future user interface would directly:

- Call repositories (save, findById, etc.), and
- Contain business rules (for example, checking if a ticket is sold, setting totalPrice, validating users).

This would:

- Mix user interface code, data access, and business logic in the same place.
- Cause duplicated logic across different parts of the program.
- Make it hard to change rules without editing many classes.

Solution (using GRASP Controller via Services):
By introducing service classes like *OrderService*, *PaymentService*, *EventService*, *TicketService*, etc., the project follows the GRASP Controller idea:

- Each service represents a controller for a specific use case (ordering, paying, managing events, etc.).
- Services coordinate repositories and models, while the user interface (added later) will only call these service methods.

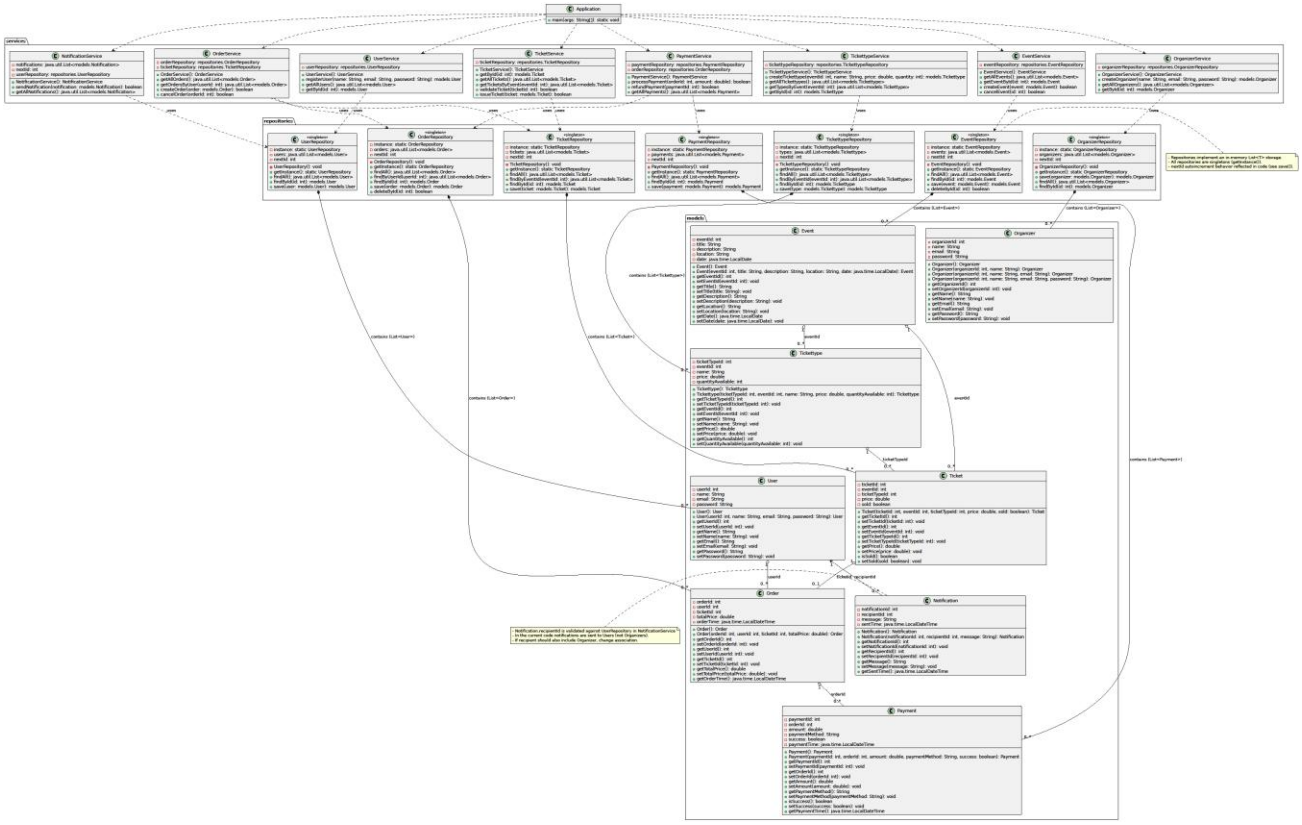
This keeps the logic centralized, easier to maintain, and ready for adding different types of UIs later without changing how the core logic works.

Conclusion

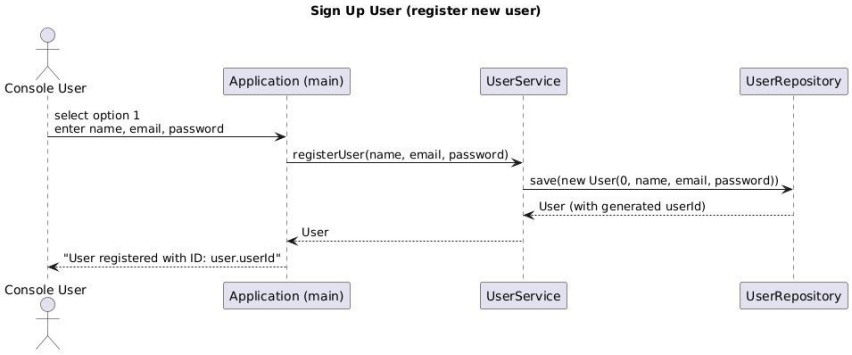
This phase established the core backend structure of the ***EventFlow application***, including domain models, repositories with ***Singleton pattern***, and service layers following ***GRASP***

principles. The implementation ensures a clear separation of concerns, data consistency, and prepares the system for future UI integration and feature expansion.

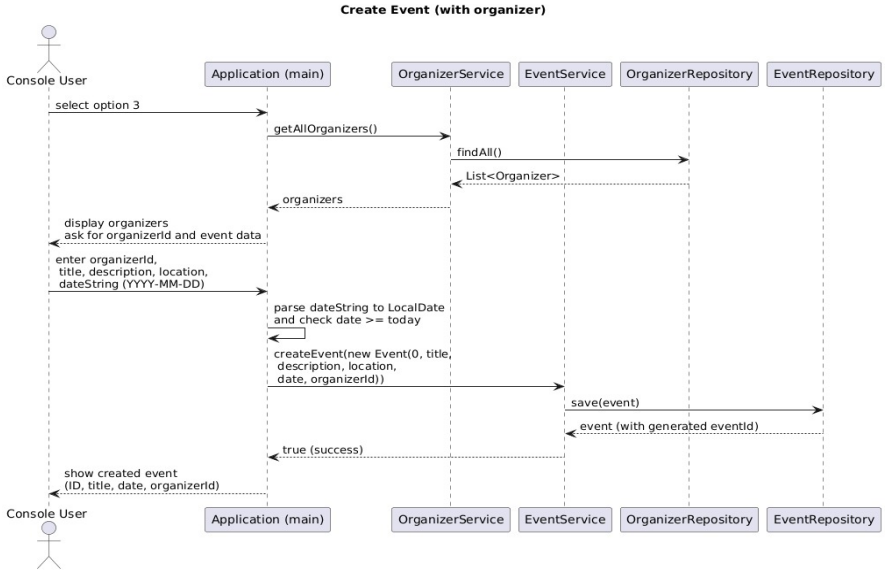
Class Diagram (also find attached in GitHub Repository):



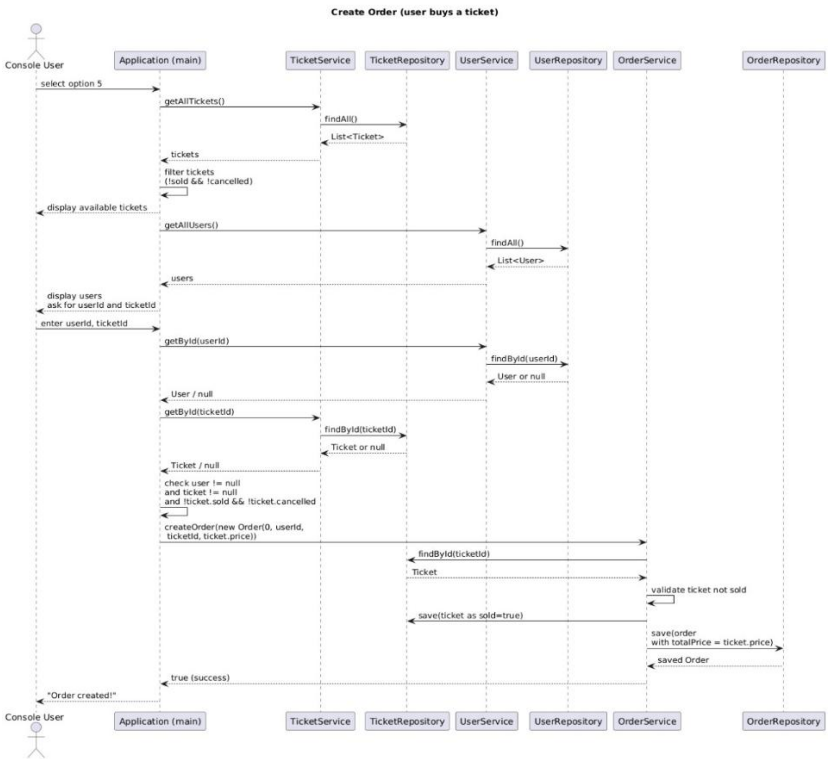
Sequence Diagram 1:



Sequence Diagram 2:



Sequence Diagram 3:



Sequence Diagram 4: