UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

Railway Management System

Submitted by:

PES1UG22CS568	**Shreevathsa G P**
PES1UG22CS571	**Shreya Kiran**
PES1UG22CS582	**Shreyas Venkatakrishnan**
PES1UG23CS837	**Vijeth Basavanakote**

Semester - 6 Section - J

Faculty Name

Dr. Bhargavi Mokashi

January - May 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FACULTY OF ENGINEERING

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road, Bengaluru – 560 085, Karnataka, India.

Problem Statement

The current railway transportation system faces significant challenges in efficiently managing train operations, ticket reservations, and passenger verification. Manual ticket booking and verification processes lead to long queues at counters, errors in reservation records, and difficulties in tracking passenger information. Furthermore, the lack of centralized management creates obstacles for administrators in monitoring system operations, managing train schedules, and generating accurate reports.

The Railway Management System aims to address these challenges by providing a comprehensive digital platform that streamlines the entire railway operation process. The system will facilitate seamless interaction between users (passengers), administrators, and ticket collectors, ensuring efficient ticket booking, train management, and passenger verification. This will reduce operational overhead, minimize human errors, enhance the passenger experience, and improve the overall efficiency of railway services.

Key Features

Admin Features:

- Train management (add, update, delete train details)
- Route management (create, modify, and delete routes)
- Schedule management (set and update train schedules)
- User management (view, block, or activate user accounts)
- Ticket collector management (assign roles, manage accounts)
- Generate reports (booking statistics, revenue, train occupancy)
- System configuration and maintenance
- Dashboard with real-time statistics

User (Passenger) Features:

- User registration and profile management
- Search available trains by various parameters (source, destination, date)
- View train details (schedule, available seats, classes, fares)
- Book tickets with seat selection
- Online payment processing
- Booking history and ticket management
- Cancel or modify existing bookings
- Receive e-tickets and booking confirmations
- Submit feedback or complaints

Ticket Collector Features:

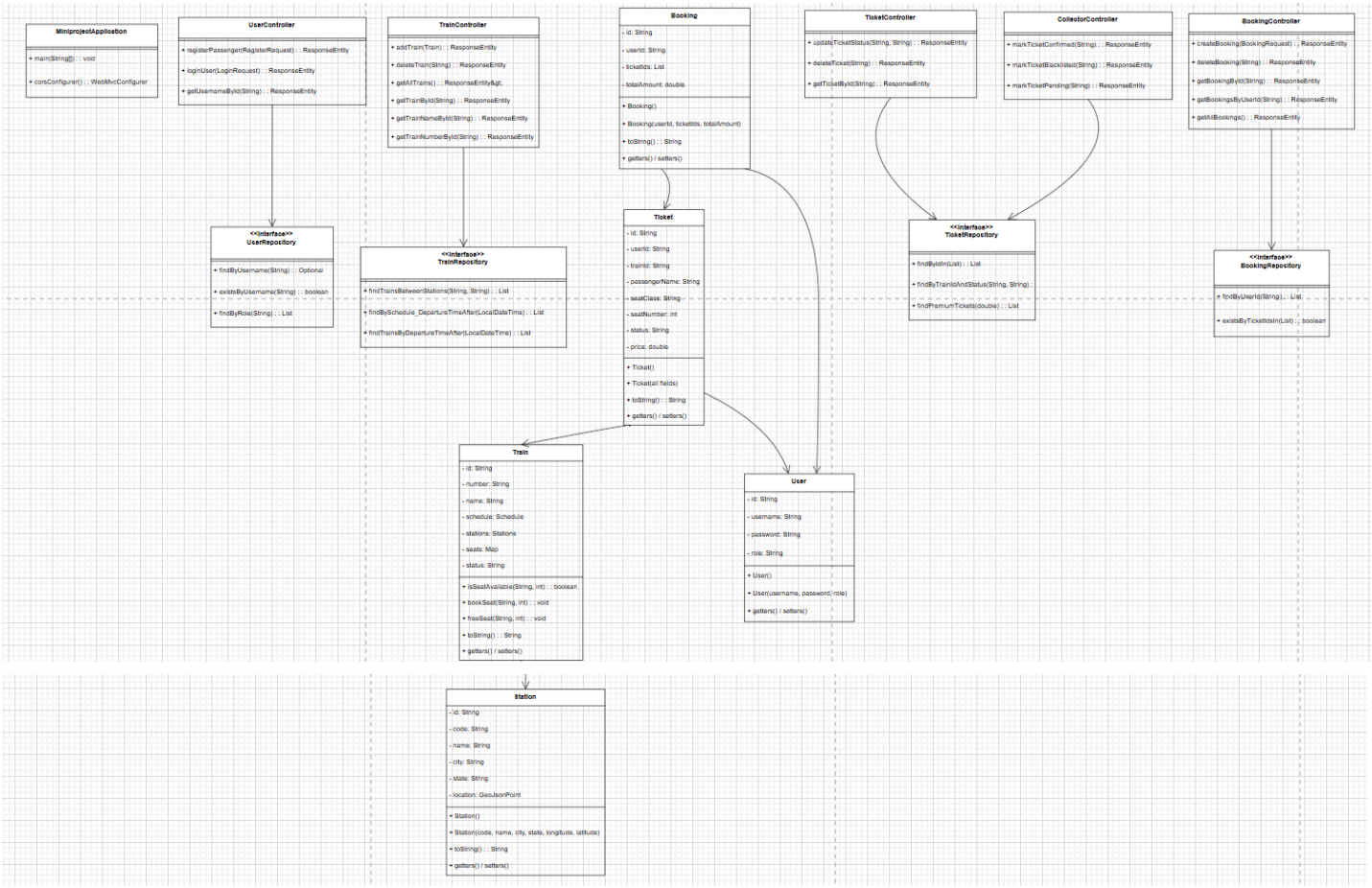
- Authentication and secure login
- View pending ticket verification requests

- Validate passenger tickets (scan/enter ticket ID)
- Blacklist fraudulent or invalid tickets
- Record verification status
- Generate verification reports
- Track passenger boarding status
- View passenger details for verified tickets

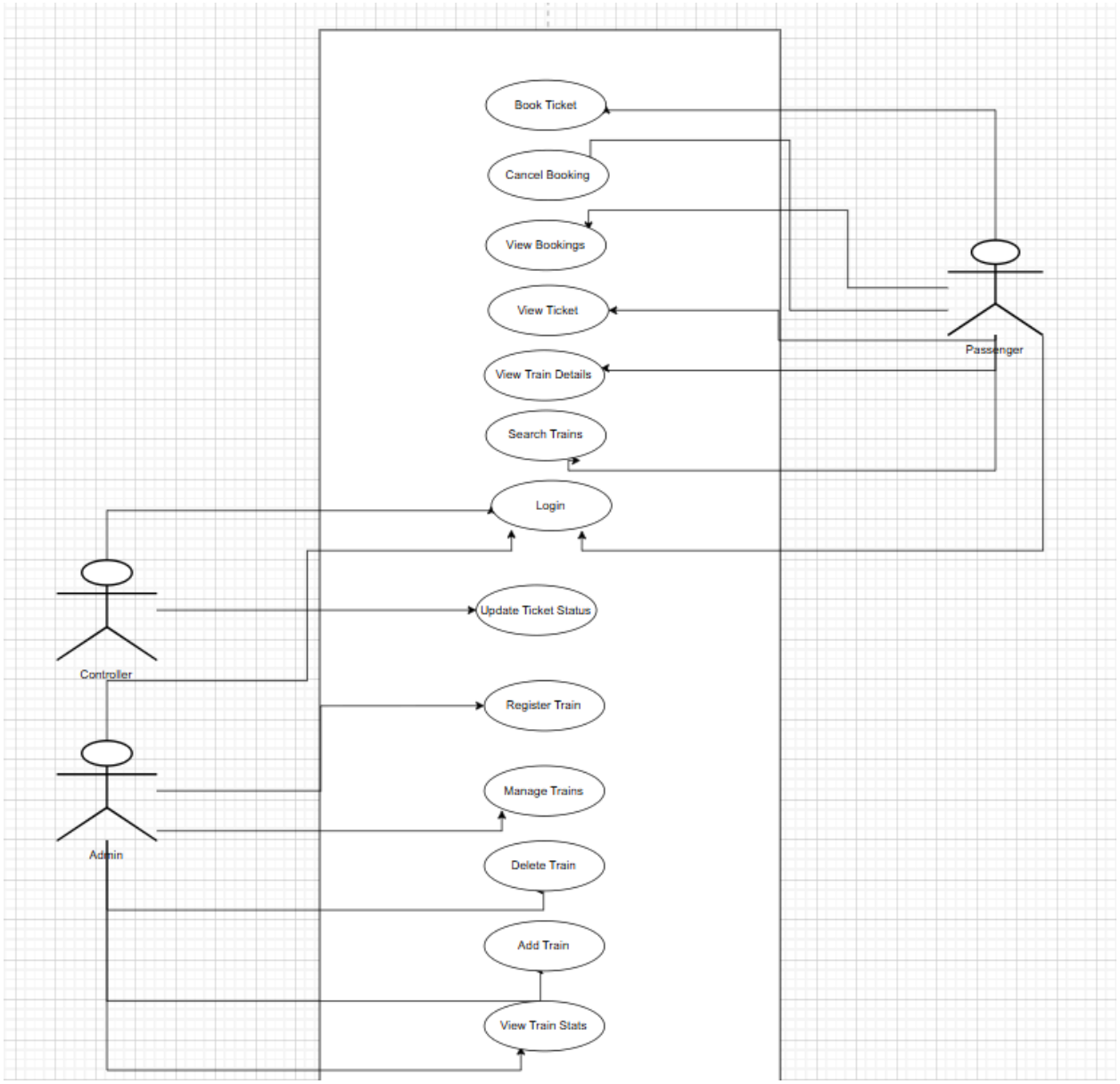
Advantages

1. **24/7 Accessibility**: Access the system anytime, anywhere, eliminating the need to visit physical booking counters.
2. **Paperless Transactions**: Environmentally friendly e-tickets reduce paper waste and eliminate the risk of lost physical tickets.
3. **Time and Cost Efficiency**: Significant reduction in queuing time and administrative costs through automation.
4. **Real-time Information**: Instant updates on train schedules, delays, and platform changes.
5. **Enhanced Security**: Secure user authentication and encrypted transactions protect personal and payment information.
6. **Centralized Data Management**: All information stored in a single database, reducing data redundancy and inconsistencies.
7. **Reduced Human Error**: Automated processes minimize errors in bookings and ticket verification.
8. **Scalability**: System designed to handle increasing user load during peak travel seasons.
9. **Integrated Analytics**: Comprehensive reporting tools for data-driven decision making.
10. **Improved Customer Experience**: User-friendly interface with intuitive navigation and responsive design.
11. **Fraud Prevention**: Ticket verification system minimizes ticket fraud and unauthorized travel.
12. **Streamlined Operations**: Automated workflows increase operational efficiency across all departments.
13. **Instant Confirmations**: Immediate booking confirmations and e-tickets delivered electronically.
14. **Transparent Pricing**: Clear fare structures with no hidden charges or intermediary fees.

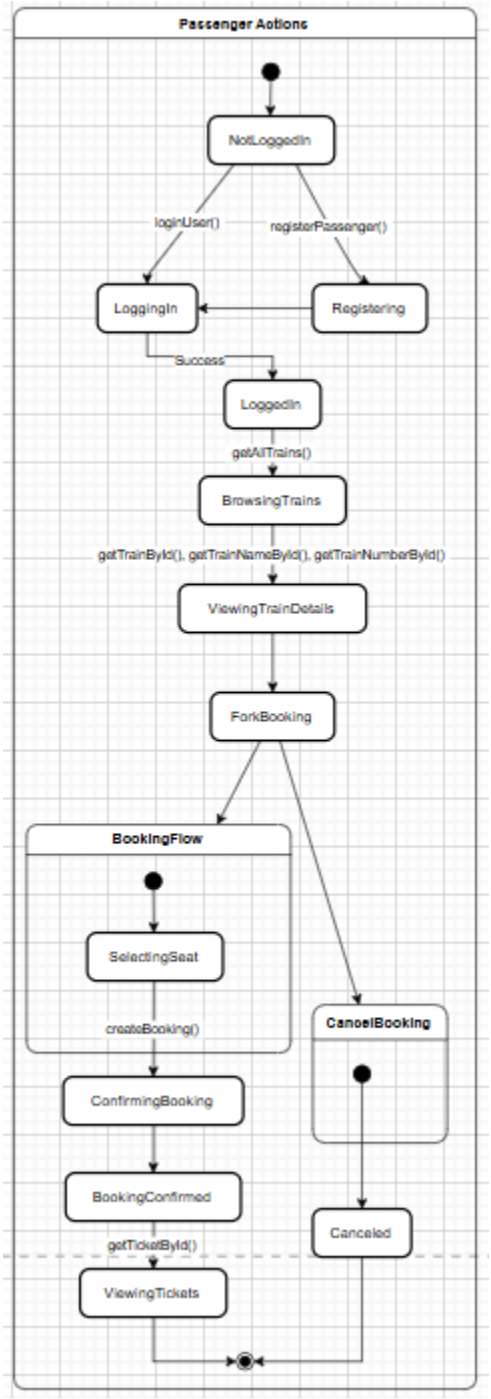
Class Diagram

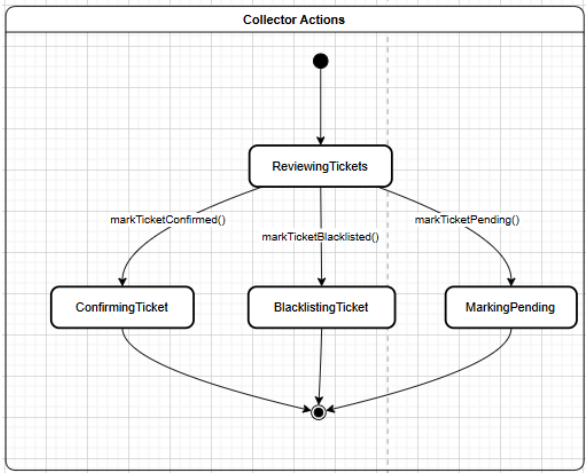


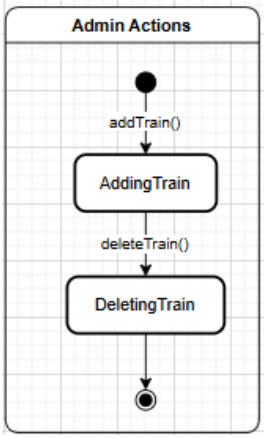
Use Case Diagram :

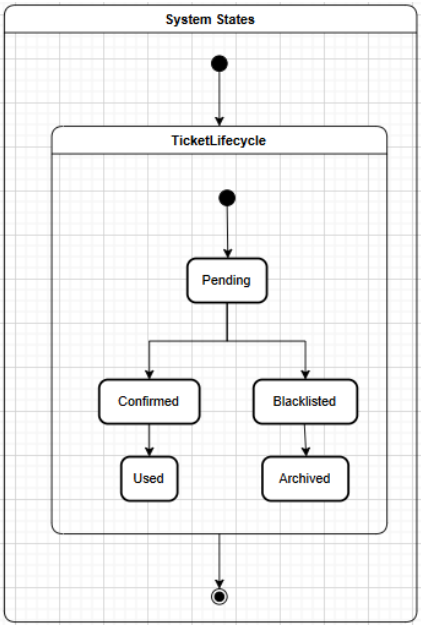


State Diagram

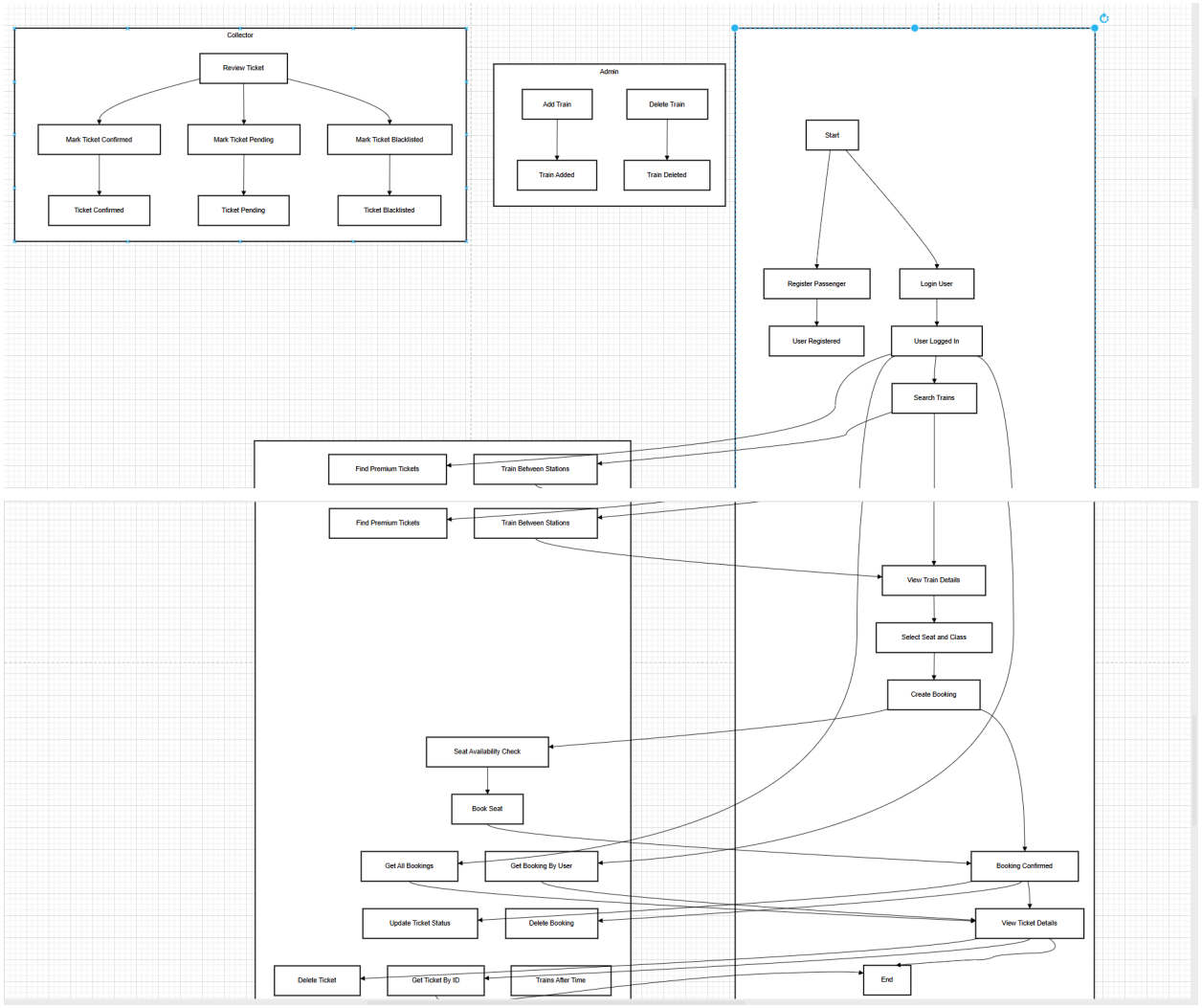








Activity Diagram



Architecture Patterns

1. **Layered Architecture (a.k.a. N-Tier Architecture)**
   Description: The layered architecture organizes the system into distinct layers, each responsible for a specific aspect of functionality. In a railway management system, this ensures clear separation between user interaction, business logic, data access, and persistence.
   Implementation in Railway Management System:
   - **Presentation Layer:** Implemented via REST controllers (e.g., UserController, TrainController, BookingController) that handle HTTP requests from clients (e.g., a React frontend or Postman). For example, POST /api/bookings creates a booking.
   - **Business Layer:** Encapsulated in service classes (e.g., BookingService, TrainService) that handle use cases like validating seat availability, calculating fares, or updating ticket

status. These services orchestrate business rules, such as ensuring a user cannot book an already reserved seat.

- **Data Access Layer:** Managed by Spring Data JPA repositories (e.g., UserRepository, TicketRepository, TrainRepository) that provide CRUD operations. For instance, TrainRepository.findById() retrieves train details for booking.
- **Persistence Layer:** Uses Spring Data JPA to interact with a relational database (e.g., MySQL or PostgreSQL), mapping entities like User, Train, and Ticket to database tables.
- **Model Layer:** Domain entities (User, Train, Ticket, Booking) represent core concepts with attributes (e.g., Train.trainId, Ticket.seatNumber) and behavior (e.g., Train.isSeatAvailable()).

  Relevance: Highly relevant, as layered architecture ensures modularity, making it easier to maintain and scale a railway system handling complex operations like ticket booking and train scheduling.

  Example: When a user books a ticket, the BookingController receives the request, BookingService validates the booking, TicketRepository saves the ticket, and the database persists the data.

2. **MVC Pattern (Model–View–Controller)**

   Description: MVC separates concerns into three components: the model (data and logic), the view (user interface), and the controller (request handling). In a REST-based railway management system, the view is often JSON responses consumed by a frontend.

   Implementation in Railway Management System:

   - **Model:** Domain classes like User, Train, Ticket, and Booking encapsulate data (e.g., Ticket.ticketId, Train.departureTime) and logic (e.g., Booking.calculateFare()).
   - **View:** Since this is a REST API, the view is the JSON response returned by controllers (e.g., ResponseEntity.ok(ticket) returns ticket details after booking). A frontend (e.g., React) could render this JSON as a ticket confirmation page.
   - **Controller:** Classes like UserController, TrainController, and BookingController handle HTTP requests. For example, TrainController.getTrainById() retrieves train details via GET /api/trains/{id}.

     Relevance: Directly applicable, as Spring Boot's REST controllers naturally follow MVC. The pattern is ideal for mapping user actions (e.g., booking a ticket) to backend logic and returning results to the frontend.

     Example: A user searches for trains via a React frontend, triggering a GET /api/trains request. TrainController processes it, retrieves data via TrainService, and returns a JSON list of trains.

3. **Repository Pattern**

   Description: The repository pattern abstracts data access, providing a collection-like interface for interacting with the database. It decouples business logic from database-specific code, improving testability and maintainability.

   Implementation in Railway Management System: Spring Data JPA interfaces (e.g., UserRepository extends JpaRepository<User, Long>, TicketRepository) define methods like findById(), save(), or custom queries (e.g., findTicketsByUserId()). These repositories handle database operations for entities like Train, Ticket, and Booking.

   Relevance: Essential for a railway system, as repositories simplify data operations (e.g.,

retrieving available trains or saving bookings) and enable unit testing by mocking repositories.
Example: BookingService uses TicketRepository.save(ticket) to persist a new ticket after a user books a seat.

4. **Dependency Injection (DI) / Inversion of Control (IoC)**
   Description: DI allows components to receive their dependencies from an external source (Spring's IoC container), reducing tight coupling and enabling easier testing and configuration.
   Implementation in Railway Management System: Spring's @Autowired annotation injects dependencies, such as UserRepository into UserService or TrainService into TrainController. The Spring container manages the lifecycle of beans (e.g., repositories, services, controllers).
   Relevance: Critical for a modular railway system, as DI allows swapping implementations (e.g., switching from MySQL to PostgreSQL) without changing service or controller code.
   Example: In BookingController, @Autowired BookingService bookingService injects the service, which itself depends on TicketRepository and TrainRepository.
5. **Domain-Driven Design (DDD)**
   Description: DDD focuses on modeling the domain based on real-world concepts, encapsulating business logic within domain entities. It emphasizes a ubiquitous language to align code with business requirements.
   Implementation in Railway Management System: Entities like Train, Ticket, and Booking represent domain concepts. For example, Train includes methods like bookSeat() and isSeatAvailable(), encapsulating train-specific logic. The system uses terms like "booking," "seat," and "train" consistently across code and requirements.
   Relevance: Highly relevant, as a railway system has a rich domain (trains, tickets, schedules) that benefits from DDD's focus on clear modeling and encapsulated logic.
   Example: The Train entity's bookSeat() method checks seat availability and updates the seat status, keeping domain logic within the entity.
6. **RESTful Architecture Style**
   Description: RESTful architecture exposes resources via stateless HTTP endpoints, using standard methods (GET, POST, PUT, DELETE) and consistent response formats. It's ideal for APIs consumed by frontends or external systems.
   Implementation in Railway Management System: Controllers expose endpoints like GET /api/trains/{id}, POST /api/bookings, and DELETE /api/tickets/{id}. ResponseEntity ensures consistent responses (e.g., 200 OK with a Ticket object or 404 Not Found). Resources (Train, Booking, Ticket) are identified by URLs.
   Relevance: Perfectly suited, as a railway system's API needs to handle CRUD operations for resources like trains and tickets, consumed by a frontend or mobile app.
   Example: A POST /api/bookings request creates a booking, returning a ResponseEntity with the booking details in JSON.

Design Principles

1. **Separation of Concerns (SoC)**
   Description: SoC divides the system into distinct modules, each handling a specific responsibility, improving maintainability and clarity.
   Implementation in Railway Management System: Controllers handle HTTP requests, services manage business logic, repositories handle data access, and entities encapsulate domain logic. For example, BookingController processes requests, while BookingService validates and processes the booking.
   Relevance: Essential for a railway system to keep concerns like user interaction, booking logic, and database access separate, enabling easier updates and debugging.
   Example: TrainController doesn't directly access the database; it delegates to TrainService, which uses TrainRepository.

2. **Single Responsibility Principle (SRP)**
   Description: SRP ensures each class has one reason to change, reducing complexity and improving maintainability.
   Implementation in Railway Management System: UserController handles user-related HTTP requests, UserService manages user-related logic (e.g., registration, authentication), and UserRepository handles user data access. Each class focuses on a single responsibility.
   Relevance: Critical, as a railway system has many components (e.g., trains, tickets, users), and SRP prevents classes from becoming bloated.
   Example: TicketService only handles ticket-related logic, such as issuing or canceling tickets, without touching train schedules.

3. **Loose Coupling**
   Description: Loose coupling minimizes direct dependencies between components, making the system more flexible and testable.
   Implementation in Railway Management System: DI ensures controllers depend on service interfaces (e.g., BookingService) rather than concrete implementations. Repositories are abstracted via interfaces, allowing database changes without affecting services.
   Relevance: Important for a railway system, as loose coupling allows replacing components (e.g., switching from JPA to MongoDB) or mocking dependencies during testing.
   Example: BookingController depends on BookingService via an interface, not a specific implementation.

4. **High Cohesion**
   Description: High cohesion groups related functionality within a module, improving clarity and reducing complexity.

Implementation in Railway Management System: The Train entity groups train-related data (e.g., trainId, seats) and methods (e.g., bookSeat()). TrainController handles all train-related endpoints, keeping related logic together.

Relevance: Relevant, as cohesive modules make it easier to understand and maintain a railway system's components (e.g., all booking logic in BookingService).

Example: BookingService contains all booking-related logic, such as validating seats and calculating fares.

5. **Open/Closed Principle (OCP)**

Description: OCP allows extending functionality without modifying existing code, ensuring stability.

Implementation in Railway Management System: New features (e.g., a discount system) can be added by creating new service methods or entities without altering existing classes. Spring's DI and interfaces support extension (e.g., adding a new repository implementation).

Relevance: Useful, as railway systems often need new features (e.g., loyalty programs) without disrupting existing booking or train management logic.

Example: A new DiscountService can be injected into BookingService to apply discounts without changing BookingService's core logic.

6. **Interface Segregation Principle (ISP)**

Description: ISP ensures clients only depend on interfaces relevant to their needs, avoiding unnecessary dependencies.

Implementation in Railway Management System: Spring Data JPA repositories define specific interfaces (e.g., TrainRepository with train-related methods). Services depend on narrow interfaces (e.g., BookingService only uses TicketRepository methods it needs).

Relevance: Applicable, as ISP prevents services or controllers from being coupled to irrelevant methods, keeping the railway system lean.

Example: BookingService uses TicketRepository.findById() but doesn't need UserRepository methods, adhering to ISP.

7. **Use of RESTful Design**

Description: RESTful design exposes resources via HTTP endpoints, following REST principles like statelessness, resource identification, and standard methods.

Implementation in Railway Management System: Controllers provide endpoints like GET /api/users/{id}, POST /api/tickets, and DELETE /api/bookings/{id}. Responses use ResponseEntity for consistency, with proper HTTP status codes (e.g., 201 Created for new bookings).

Relevance: Central to the system, as RESTful APIs enable a frontend or mobile app to interact with the railway system's backend.

Example: POST /api/tickets creates a ticket, returning a ResponseEntity with the ticket's details and a 201 status.

8. **KISS (Keep It Simple, Stupid)**
   Description: KISS advocates simplicity in design and implementation, avoiding unnecessary complexity.
   Implementation in Railway Management System: The system uses straightforward REST endpoints, simple domain models, and minimal logic in controllers. For example, TrainController delegates complex logic to TrainService rather than handling it directly.
   Relevance: Important, as a railway system should be easy to understand and maintain, especially for critical operations like booking and scheduling.
   Example: The Ticket entity has simple attributes (ticketId, seatNumber) and methods (cancelTicket()), avoiding overcomplication.

9. **Encapsulation**
   Description: Encapsulation hides internal details of a class, exposing only necessary interfaces and protecting data integrity.
   Implementation in Railway Management System: Domain models like Train encapsulate data (e.g., seats) and behavior (e.g., bookSeat()), using private fields and public methods. For example, Train.bookSeat() ensures seats are only booked if available.
   Relevance: Critical, as encapsulation ensures domain logic (e.g., seat validation) is protected and consistent in a railway system.
   Example: The Booking entity encapsulates calculateFare(), ensuring fare logic is internal and not manipulated externally.

Design Patterns

1. **Model-View-Controller (MVC) Pattern**
   Description: MVC separates the application into three components: the model (data and logic), the view (user interface), and the controller (request handling). In a REST API, the view is typically JSON responses.
   Implementation in Railway Management System:
   - Model: User, Train, Ticket, and Booking classes hold data and logic (e.g., Train.isSeatAvailable()).
   - View: JSON responses from controllers (e.g., ResponseEntity.ok(booking)) serve as the view, consumed by a React frontend or other clients.
   - Controller: UserController, TrainController, and BookingController handle HTTP requests, delegating to services.
     Relevance: Fully relevant, as Spring Boot's REST architecture inherently uses MVC,

making it ideal for a railway system's API.
Example: A POST /api/bookings request is handled by BookingController, which uses BookingService to create a booking and returns a JSON response.

2. **Repository Pattern**
   Description: The repository pattern provides a collection-like interface for data access, abstracting database operations and improving testability.
   Implementation in Railway Management System: Spring Data JPA repositories (UserRepository, TrainRepository, TicketRepository) define methods like save(), findById(), or custom queries (e.g., findTrainsByRoute()). These handle database interactions for entities.
   Relevance: Highly relevant, as repositories simplify data access for a railway system's entities and enable mocking for testing.
   Example: BookingService uses TicketRepository.save(ticket) to persist a ticket after booking.

3. **Template Method Pattern**
   Description: The template method pattern defines a skeleton for an algorithm, allowing subclasses to customize specific steps. Spring Data JPA/MongoDB uses this internally to standardize database operations.
   Implementation in Railway Management System: Spring Data JPA's repository interfaces (e.g., JpaRepository) provide predefined methods (save(), findAll()) that follow a template. Custom queries (e.g., findTicketsByUserId()) extend this template.
   Relevance: Relevant but less explicit, as the pattern is used internally by Spring Data, not directly implemented in the application code. It simplifies repository usage in the railway system.
   Example: TrainRepository.findAll() follows Spring's template for querying trains, while findByRoute() customizes the query.

4. **Dependency Injection (DI)**
   Description: DI manages object dependencies by injecting them externally, promoting loose coupling and testability.
   Implementation in Railway Management System: Spring's @Autowired injects dependencies like BookingService into BookingController or TrainRepository into TrainService. The @Bean annotation in MiniprojectApplication defines custom beans (e.g., WebMvcConfigurer).
   Relevance: Essential, as DI enables modular development and testing in a railway system with many interconnected components.
   Example: TrainController uses @Autowired TrainService trainService to access train-related logic.

5. **Builder Pattern**

   Description: The builder pattern constructs complex objects step by step, improving readability and flexibility.

   Implementation in Railway Management System: The Train class's nested classes (Schedule, SeatClass) act like builders, encapsulating related data (e.g., departure time, seat type) and providing methods to set properties. Alternatively, a BookingBuilder could be used to construct Booking objects with optional fields (e.g., discounts, preferences).

   Relevance: Moderately relevant. While nested classes mimic the builder pattern, a dedicated Builder class (e.g., Booking.builder().withUser(user).withTrain(train).build()) is not explicitly mentioned. If implemented, it would simplify creating complex Booking or Train objects.

   Example: A hypothetical BookingBuilder could create a Booking with new BookingBuilder().setUser(user).setTrain(train).setSeat(seat).build().

6. **Observer Pattern**

   Description: The observer pattern allows objects to subscribe to and react to events or state changes in another object. In a React frontend, this is used to update the UI based on state changes.

   Implementation in Railway Management System: The backend (Java/Spring) does not explicitly use the observer pattern. However, the React frontend uses useState and useEffect hooks to observe state changes (e.g., a list of available trains) and update the UI. For example, a train search result triggers a UI update when the API response arrives.

   Relevance: Relevant for the frontend but not the Java backend. In a railway system, the backend could use the observer pattern (e.g., via Spring's @EventListener) for scenarios like notifying users of train delays, but this isn't mentioned in your context.

   Example: In the React frontend, useEffect fetches trains via GET /api/trains and updates the UI when the response changes the trains state.