**Introduction**

This guide provides a detailed breakdown for implementing the Real-Time Event Ticketing System, focusing on the use of Object-Oriented Programming (OOP) principles and the Producer-Consumer pattern to simulate a dynamic ticketing environment. The system will handle concurrent ticket releases and purchases while maintaining data integrity.

**Project Requirements**

**Objective**: Build a real-time ticketing system that concurrently manages ticket releases by vendors (producers) and purchases by customers (consumers), ensuring thread safety and efficient synchronization.

**Technology Stack**:

* **CLI**: Java
* **Frontend**: JavaFX, React.js, or Angular
* **Backend**: Java (for JavaFX integration), Node.js, or Spring Boot

**1. Java CLI: Core Java Implementation**

The CLI component will serve as the primary user interface for configuring and controlling the system. This section emphasizes core Java OOP principles and requires a command-line setup, configuration, and operational monitoring.

**Requirements:**

* **System Configuration**:
  + Implement a command-line-based configuration process to set system parameters:
    - **Total Tickets (totalTickets)**: The total tickets available in the system.
    - **Ticket Release Rate (ticketReleaseRate)**: How frequently vendors add tickets.
    - **Customer Retrieval Rate (customerRetrievalRate)**: How often customers purchase tickets.
    - **Max Ticket Capacity (maxTicketCapacity)**: Maximum capacity of tickets the system can hold.
  + **Input Validation**:
    - Validate user inputs, ensuring only acceptable values are entered for parameters.
    - Re-prompt for input if values are invalid.
* **Command Execution**:
  + **Start/Stop Commands**: Commands to initiate and terminate ticket handling operations.
  + **Real-Time Monitoring**:
    - Show real-time ticket pool status (e.g., tickets being added or removed).
    - Log all ticket sales and additions to keep a record of transactions.
* **Implementation of Producer-Consumer Pattern**:
  + Use core Java multi-threading and synchronization mechanisms (e.g., synchronized blocks, locks) to implement the producer-consumer pattern.
  + Simulate multiple vendors and customers using Java’s Runnable interface to manage threads safely.

**Deliverables for CLI:**

1. **Configuration Setup**: Console prompts for input with error-handling.
2. **Execution and Logging**: Real-time ticket updates in the console.
3. **Documentation**: In-line comments, configuration instructions, and basic troubleshooting guidance.

**2. Frontend GUI: JavaFX, React.js, or Angular**

The GUI will provide a more user-friendly interface for non-technical users to interact with the system. The goal is to demonstrate the ability to build a robust interface that reflects real-time data.

**Requirements:**

* **Interface Design**:
  + **Display Sections**:
    - **Ticket Pool Status**: Real-time view of ticket availability.
    - **Control Panel**: Start, stop, and reset buttons.
    - **Configuration Settings**: Fields for setting or adjusting parameters before starting the system.
  + **Error Handling UI**:
    - Display notifications or warnings for invalid entries.
    - Real-time status updates and error messages for ease of monitoring.
* **GUI Components**:
  + **JavaFX (if chosen)**:
    - Use VBox, HBox, or GridPane for layout design.
    - Include interactive elements such as TextField, Button, and Label.
  + **React.js or Angular (if chosen)**:
    - Components like ConfigurationForm, TicketStatus, ControlPanel, and LogDisplay to organize functionality.
    - Use state management (e.g., useState in React, RxJS in Angular) to handle real-time updates from the backend.
* **Communication with Backend**:
  + **Polling or WebSockets**:
    - Establish a way to fetch updates from the backend (WebSocket preferred for real-time updates or periodic polling for simple implementations).

**Deliverables for GUI:**

1. **Real-Time Data Display**: Reflects ticket pool status, controls, and configuration options.
2. **User Interactivity**: Valid inputs and responsive control buttons.
3. **Frontend Documentation**: Details for setting up and using the GUI interface.

**3. Backend: Multi-Threaded Producer-Consumer Logic**

The backend handles core functionalities, enforcing concurrency and synchronization for vendors and customers as they interact with the ticket pool. This section should focus on thread management and data integrity.

**Requirements:**

* **Vendor and Customer Threads**:
  + Implement multi-threaded classes for vendors and customers, each represented as separate threads.
  + **Producer-Consumer Synchronization**:
    - Ensure that multiple vendors (producers) and customers (consumers) interact safely with the shared TicketPool.
  + **Concurrency Control**:
    - Use synchronization techniques (synchronized, ReentrantLock) to ensure thread safety and prevent race conditions.
* **TicketPool Class**:
  + Implement a shared resource class to manage tickets.
  + **Data Structure**:
    - Use a thread-safe data structure (e.g., Collections.synchronizedList, ConcurrentLinkedQueue) to store tickets.
  + **Methods**:
    - addTickets(): Allows vendors to add tickets to the pool.
    - removeTicket(): Allows customers to purchase tickets, reducing the ticket count.
    - **Capacity Checks**: Ensure maxTicketCapacity is never exceeded by producers.
* **Backend Technologies**:
  + **Java (if integrated with JavaFX)**:
    - Implement the backend logic directly using core Java.
  + **Node.js or Spring Boot (if chosen)**:
    - Use Node.js or Spring Boot to manage the backend if building a web-based frontend.
    - Implement RESTful endpoints for GUI interaction if required.

**Deliverables for Backend:**

1. **Multi-Threaded Producer-Consumer Implementation**: Complete synchronization and threading for handling ticket additions and purchases.
2. **Ticket Management and Concurrency**: Thread-safe ticket pool with synchronized methods.
3. **Backend Documentation**: Explanation of threading, synchronization choices, and troubleshooting tips.

**Optional Advanced Functionalities (Bonus Marks)**

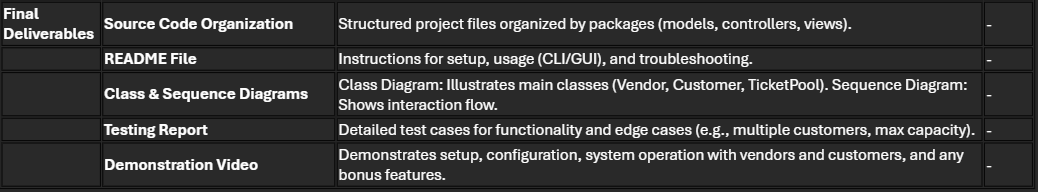
1. **Priority Customers**:
   * Implement a VIP customer class that has higher priority access to tickets.
   * Use a priority queue or a custom mechanism to allow VIP customers access before regular customers.
2. **Dynamic Vendor/Customer Management**:
   * Allow adding or removing vendors/customers at runtime through the GUI or CLI.
3. **Real-Time Analytics Dashboard**:
   * Visual charts to display ticket sales over time using charting libraries.
4. **Advanced Synchronization**:
   * Use ReentrantLock or Semaphore for more fine-grained control over shared resources.
5. **Persistence**:
   * Store transaction data in a simple database (e.g., SQLite) to track ticket sales and customer data.

**Final Deliverables**

1. **Source Code**:
   * Well-structured project files with organized packages (e.g., models, controllers).
2. **README File**:
   * Setup instructions, CLI and GUI usage guidelines, troubleshooting information.
3. **Diagrams**:
   * **Class Diagram**: Show main classes (Vendor, Customer, TicketPool).
   * **Sequence Diagram**: Illustrate interactions for ticket purchase and release processes.
4. **Testing Report**:
   * Include scenarios tested (e.g., handling multiple customers, max capacity breaches).
5. **Demonstration Video**:
   * Show setup, configuration, system running with multiple vendors and customers, and any advanced features implemented.

**Implementation Timeline and Tips**

1. **Begin with CLI**:
   * Start with the CLI as it provides the foundation for backend configuration and control.
2. **Move to Backend Logic**:
   * Implement core producer-consumer threading and synchronization in the backend.
3. **Complete Frontend GUI**:
   * Build the GUI last, ensuring it can properly display backend data and handle user inputs.

A screenshot of a computer

Description automatically generatedA screenshot of a black screen

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated