ĐẠI HỌC KINH TẾ THÀNH PHỐ HỒ CHÍ MINH
VIỆN ĐỔI MỚI SÁNG TẠO

BÁO CÁO ĐỒ ÁN MÔN HỌC LẬP TRÌNH ỨNG DỤNG

Đề tài: ỨNG DỤNG QUẢN LÝ NHÀ HÀNG

Giảng viên hướng dẫn: ThS. Huỳnh Viết Thám

Nhóm sinh viên thực hiện: Nhất xạ tứ nô

Họ và tên sinh viên:

1. Đào Nguyễn Phúc Quang
2. Trần Khôi Nguyên
3. Lê Huỳnh Thảo Phương
4. Nguyễn Quang Phúc
5. Trần Anh Vũ

TP. Hồ Chí Minh, ngày 24 tháng 12 năm 2025

Mục lục

Danh sách hình vẽ

Chương 1 Giới thiệu

1.1 Giới thiệu đề tài

Trong bối cảnh chuyển đổi số đang diễn ra mạnh mẽ trên toàn cầu, việc ứng dụng công nghệ thông tin vào quản lý kinh doanh trở thành một xu thế tất yếu, đặc biệt đối với các cửa hàng bán lẻ vừa và nhỏ. Hoạt động kinh doanh trong lĩnh ngành dịch vụ ăn uống (F&B) đang đối mặt với những thách thức lớn về tối ưu hóa vận hành và nâng cao trải nghiệm khách hàng. Để tồn tại và phát triển, các chủ nhà hàng không chỉ cần món ăn ngon, dịch vụ tốt, mà còn phải quản lý hiệu quả mọi khía cạnh hoạt động: từ nhân sự, ca làm việc, kho nguyên liệu, đến việc sắp xếp đặt bàn cho khách hàng.

Tuy nhiên, thực tế hiện nay tại nhiều nhà hàng quy mô vừa và nhỏ ở Việt Nam vẫn đang phụ thuộc vào các phương pháp quản lý thủ công: ghi chép bằng sổ tay, bảng Excel, hoặc các phần mềm rời rạc không liên kết với nhau. Cách làm này thường dẫn đến sai sót, mất thời gian, khó theo dõi tồn kho, lịch làm việc chồng chéo, hoặc tình trạng khách hàng đặt bàn nhưng đến nơi lại không có chỗ – những vấn đề trực tiếp ảnh hưởng đến doanh thu và uy tín của nhà hàng. Những thách thức trong việc kiểm soát hàng hóa, đơn hàng, nhân sự và tài chính khiến cho các chủ cửa hàng ngày càng cần đến những giải pháp phần mềm hiệu quả, dễ sử dụng và tiết kiệm chi phí.

Xuất phát từ thực trạng trên, nhóm đã chọn phát triển **Ứng dụng quản lý nhà hàng (Restaurant Management System)** – một giải pháp phần mềm desktop toàn diện, được thiết kế dành riêng cho các nhà hàng quy mô vừa và nhỏ, phù hợp với môi trường làm việc offline, dễ triển khai và không đòi hỏi hạ tầng công nghệ phức tạp. Ứng dụng được xây dựng bằng ngôn ngữ lập trình Java, sử dụng JavaFX để tạo giao diện người dùng hiện đại, trực quan và áp dụng các nguyên lý lập trình hướng đối tượng (OOP) cùng kiến trúc phân lớp. Các chức năng chính bao gồm:

- Quản lý thông tin nhân viên và ca làm việc.
- Quản lý kho hàng và các giao dịch liên quan đến nhập – xuất kho.
- Quản lý đặt bàn/đơn hàng của khách hàng.

- Hỗ trợ tìm kiếm, lọc và hiển thị danh sách dữ liệu.
- Lưu trữ dữ liệu trong bộ nhớ thông qua các lớp repository.

1.2 Mục tiêu của dự án

1.2.1 Mục tiêu tổng quát

Xây dựng một ứng dụng quản lý nhà hàng hoàn chỉnh, có khả năng đáp ứng và chuyển đổi phương thức quản lý từ truyền thống sang số hóa các nghiệp vụ quản lý cơ bản, hoạt động ổn định, giao diện thân thiện và dễ sử dụng. Ứng dụng hướng đến việc hỗ trợ người quản lý giảm bớt khối lượng công việc thủ công, nâng cao hiệu quả quản lý, kiểm soát toàn diện hoạt động kinh doanh, từ khâu bán hàng đến quản lý nhân sự và tài chính và hạn chế sai sót trong quá trình vận hành.

1.2.2 Mục tiêu cụ thể

- Thiết kế và triển khai hệ thống theo kiến trúc phân lớp (layered architecture) chuyên nghiệp, bao gồm các tầng: Model, Repository, Service, Controller và View, đảm bảo tính dễ bảo trì, mở rộng và tái sử dụng mã nguồn.
- Áp dụng triệt để các nguyên lý lập trình hướng đối tượng (OOP): đóng gói (encapsulation), kế thừa (inheritance), đa hình (polymorphism), trừu tượng (abstraction), đồng thời sử dụng các design pattern phù hợp (như Singleton, Factory, Observer).
- Xây dựng giao diện người dùng đẹp mắt, trực quan bằng JavaFX, hỗ trợ các thành phần hiện đại như TableView, DatePicker, ComboBox, ObservableList, v.v.
- Hoàn thiện đầy đủ các chức năng CRUD (Create – Read – Update – Delete) cho các thực thể chính: nhân viên, ca làm việc, nguyên vật liệu, giao dịch kho, đặt bàn.

1.3 Phạm vi của dự án

1.3.1 Các chức năng được triển khai.

Trong phạm vi của dự án, nhóm tập trung triển khai các chức năng quản lý cơ bản và thiết yếu cho một cửa hàng/nhà hàng, bao gồm:

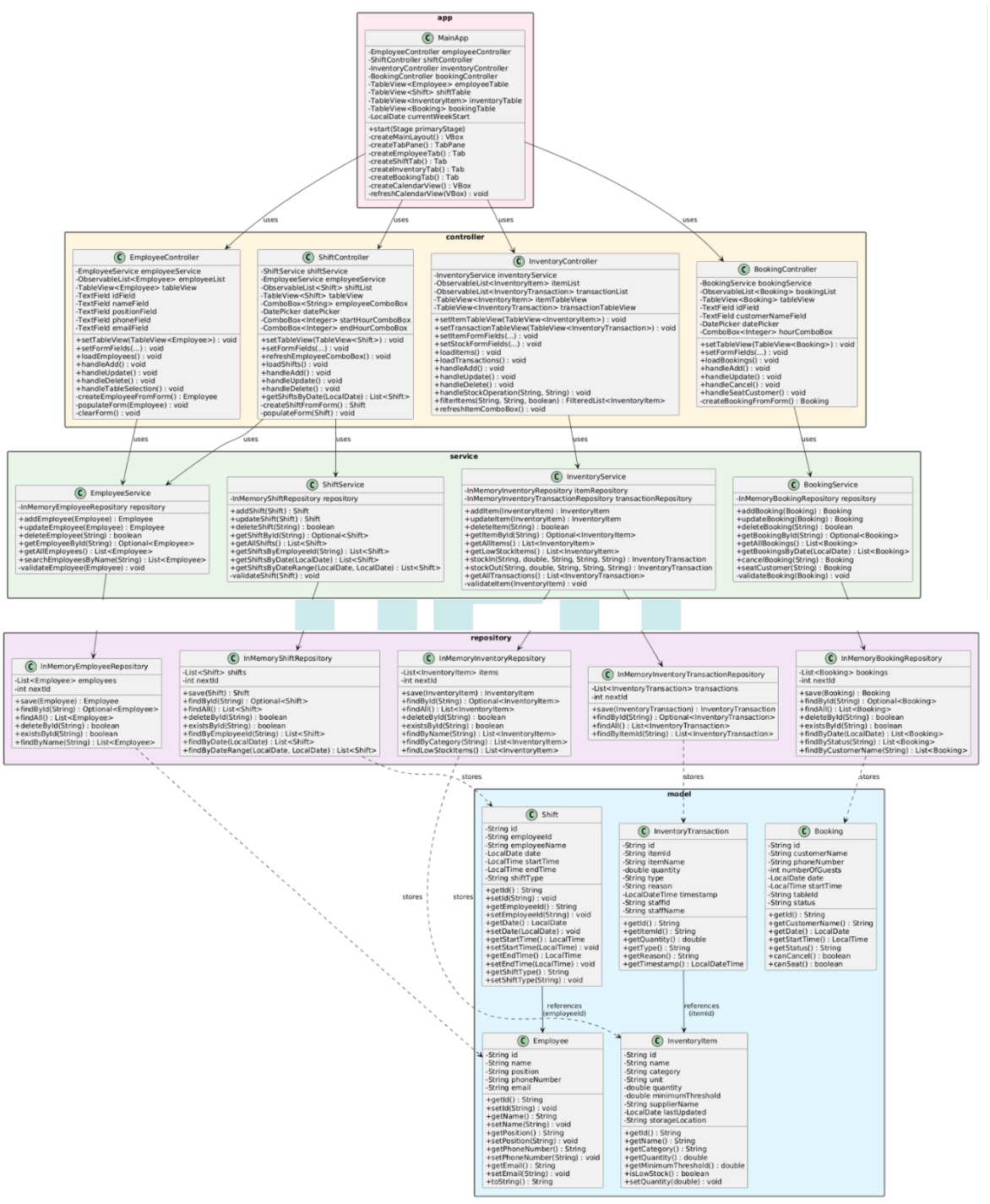
- **Quản lý nhân viên:** lưu trữ và cập nhật thông tin nhân viên như họ tên, chức vụ, số điện thoại, email; cập nhật và xóa dữ liệu khi cần thiết.
- **Quản lý ca làm việc:** phân công ca làm theo ngày, theo nhân viên; theo dõi thời gian bắt đầu và kết thúc ca làm việc, kiểm tra xung đột lịch trình và theo dõi bảng lương.
- **Quản lý kho hàng:** theo dõi danh sách nguyên liệu (tên, đơn vị, số lượng tồn kho, ngưỡng tồn kho tối thiểu), thực hiện nhập kho/xuất kho, ghi nhận lịch sử giao dịch kho, và số lượng tồn kho, ngưỡng tồn kho tối thiểu.
- **Quản lý đặt bàn/đơn hàng:** cho phép khách hàng đặt bàn trước (tên khách, số điện thoại, số lượng người, ngày giờ), hủy đặt bàn, xem danh sách đặt bàn theo ngày và trạng thái (đã xác nhận, đang chờ, đã hủy).
- **Hỗ trợ tìm kiếm và lọc dữ liệu:** giúp người quản lý dễ dàng tra cứu thông tin theo nhiều tiêu chí khác nhau.

1.3.2 Các chức năng không được triển khai.

Do giới hạn về thời gian và phạm vi của đồ án, một số chức năng nâng cao chưa được triển khai trong hệ thống, bao gồm:

- Kết nối và lưu trữ dữ liệu bằng hệ quản trị cơ sở dữ liệu (MySQL, SQL Server,…).
- Phân quyền người dùng và cơ chế bảo mật đăng nhập nâng cao.
- Thống kê, báo cáo doanh thu và hiệu suất hoạt động dưới dạng biểu đồ.
- Tích hợp thanh toán trực tuyến hoặc các dịch vụ bên ngoài.
- Triển khai hệ thống trên môi trường web hoặc mobile.

Chương 2 Phân tích và thiết kế hệ thống



Hình 2-1. Sơ đồ UML

2.1 Sơ đồ UML:

Tổng quan của UML: Hệ thống được thiết lập phân tầng rõ ràng, chặt chẽ. Dữ liệu được chảy một chiều từ trên xuống dưới: UI → Controller → Service →

Repository → Model. Mỗi tầng trong sơ đồ chỉ làm đúng 1 vai trò, không làm thay việc của các tầng khác. Ngoài ra, các tầng chỉ làm việc với tầng kế bên nó.

Dòng chảy của dữ liệu bắt đầu khi người dùng tương tác với giao diện người dùng (UI). Khi này, UI không hề truy cập vào cơ sở dữ liệu mà chỉ đóng vai trò thu nhận dữ liệu và gửi nó cho Controller. Sau đó, tầng này sẽ kiểm tra định dạng dữ liệu, chuyển đổi dữ liệu giữa UI và model và điều phối việc load, hiển thị dữ liệu và gọi các nghiệp vụ. Dòng DB được điều phối vào từng Service theo . Đây là nơi xử lý chính và kiểm tra các logic, điều kiện ràng buộc của dữ liệu. Tuy nhiên, tầng này không trực tiếp lưu trữ, truy cập và xử lý dữ liệu mà truyền xuống cho Repository. Đây là tầng chịu trách nhiệm truy cập dữ liệu khi Service gửi yêu cầu và trả lại kết quả. Repository không chứa logic của function mà chỉ đọc và ghi dữ liệu. Cuối cùng, tầng Model đại diện cho đối tượng mà hệ thống nghiệp vụ đang làm việc. Nó chỉ có chức năng lưu trữ thông tin, hành vi của đối tượng nghiệp vụ và biểu diễn trạng thái hiện tại của nó. Các đối tượng trong tầng Model được ánh xạ từ dữ liệu mà Repository cập nhật.

Sau khi đã có được đối tượng trong Model, nó sẽ được trả lên ngược lại Service, Controller để kiểm tra, chuẩn hóa lại đối tượng và điều phối các phản hồi phù hợp cho giao diện người dùng. Cuối cùng, UI sẽ trả về đối tượng được Controller điều phối tương ứng với dữ liệu đầu vào, hoàn tất luồng trả kết quả.

Hướng thiết kế của UML: Sử dụng mô hình MVC (Model-View-Controller) của JavaFX để hiển thị và xử lý sự kiện.

Chi tiết từng tầng UML:

1. **Tầng Model:** Cách để đọc dữ liệu của tầng này để truyền lên những tầng trên là dùng chuẩn hóa dữ liệu, vì các giá trị người dùng nhập sẽ được tham chiếu chuẩn hoá dưới dạng ID được khởi tạo từ máy. Ví dụ như class Shift sẽ lấy employeeID thay vì trực tiếp tham chiếu các dữ liệu của Employee đó. Chuẩn hóa dữ liệu giúp ứng dụng chạy nhanh hơn do không phải load quá nhiều dữ liệu mà chỉ cần tham chiếu đến ID cần dùng.

- **Class Employee:** Gồm id, name, position, phoneNumber, email đều là String, trong đó quan trọng nhất là id làm Primary Key để tra cứu thông tin của Employee nhanh hơn mà không tốn nhiều bộ nhớ.
- **Class Shift:** Gồm id, employeeId, employeeName, date, startTime, endTime, shiftType, mỗi String đều có chức năng chính của riêng String đó, trong đó id là quan trọng đóng vai trò định danh điểm dữ liệu.
- **Class InventoryItem:** Gồm id, name, category, unit, quantity, minimumThreshold, supplierName, lastUpdated, storageLocation. MinimumThreshold sẽ có chức năng kiểm tra xem quantity có bé hơn giá trị minimumThreshold không, nếu có thì trả về Low stock cảnh báo người quản lý.
- **Class InventoryTransaction (Lịch sử kho):** Gồm id, itemId, itemName, quantity, type, reason, timestamp, staffID, staffname.
- **Class Booking:** Gồm id, customerName, phongNumber, numberOfGuests, date, startTime, tableId, status. Trong đó, status chỉ trạng thái của bàn được đặt (pending, seated, cancelled) và người dùng có thể điều chỉnh trạng thái của bàn.

2. **Tầng Repository:** Tầng này dùng để lưu trữ thông tin của người dùng nhập vào nhưng không tiết lộ chi tiết về việc lưu ở đâu. Cách lưu dữ liệu của tầng này là dùng ID của các điểm dữ liệu, sử dụng static int nextID để gợi ý cơ chế tự tăng ID thủ công khi người dùng nhập vào một điểm dữ liệu mới. List<tên Class> được dùng làm kho chứa dữ liệu.

   Các class InMemory… mô phỏng các Database:

   - **Items/employees/shifts(List<...>):** Đóng vai trò như một bảng Database, nó lưu dữ liệu trong RAM và sẽ mất khi tắt ứng dụng hoặc sập nguồn.
   - **nextId (int):** Đóng vai trò làm biến đếm, khi tạo một bản ghi mới thì biến này tự tăng thêm 1 và gán vào Id mới, đảm bảo không trùng lặp ID

3. **Tầng Service (tầng mà chương trình hoạt động):** Đây là tầng chứa các phương thức để xử lý các dữ liệu cũng như thao tác của người dùng.
   - **Logic chung:** Các class Service đều sẽ chứa một thuộc tính repository, giúp tầng này không cần biết dữ liệu được lưu ở đâu, chỉ cần gọi repository.save() hoặc repository.findAll().
   - **InventoryService:** trong class này có itemRepository để cập nhật số lượng tồn kho của hàng, transactionRepository để ghi lại nhật ký nhập/xuất của món hàng đó. Class này sẽ giúp làm 2 nhiệm vụ này một lúc trên 2 bảng dữ liệu khác nhau.
4. **Tầng Controller:** Các thuộc tính ở đây chủ yếu là Dependencies (Phụ thuộc) và UI Bindings (Kết nối giao diện)
   - **Class EmployeeController:**
     - **employeeService:** Gọi logic nghiệp vụ.
     - **employeeList (ObservableList<Employee>):** Danh sách nhân viên có khả năng tự động cập nhật UI. Khi list này thay đổi, TableView tự vẽ lại.
     - **tableView:** Tham chiếu đến bảng hiển thị trên màn hình.
     - **idField, nameField, ... (TextField):** Các ô nhập liệu trên Form. Controller lấy dữ liệu từ đây khi user bấm "Save".
   - **Class ShiftController**
     - **shiftService:** Logic ca làm.
     - **employeeService:** Cần thiết để load danh sách nhân viên vào ComboBox cho người dùng chọn khi tạo ca.
     - **shiftList**: Dữ liệu cho bảng ca làm.
     - **employeeComboBox:** Dropdown chọn nhân viên.
     - **startHourComboBox, endHourComboBox:** Dropdown chọn giờ (đơn giản hóa việc nhập giờ).
   - **Class InventoryController**
     - **inventoryService:** Logic kho.
     - **itemList:** Dữ liệu tồn kho hiện tại.
     - **transactionList:** Dữ liệu lịch sử giao dịch.

- **itemTableView:** Bảng hiển thị hàng hóa.
- **transactionTableView:** Bảng hiển thị lịch sử (thường nằm ở tab khác).

– **Class BookingController**

- **bookingService:** Logic đặt bàn.
- **bookingList:** Dữ liệu đặt bàn.
- **hourComboBox:** Chọn giờ đặt.

5. **Tầng app (Ứng dụng):**

– **Class MainApp:**

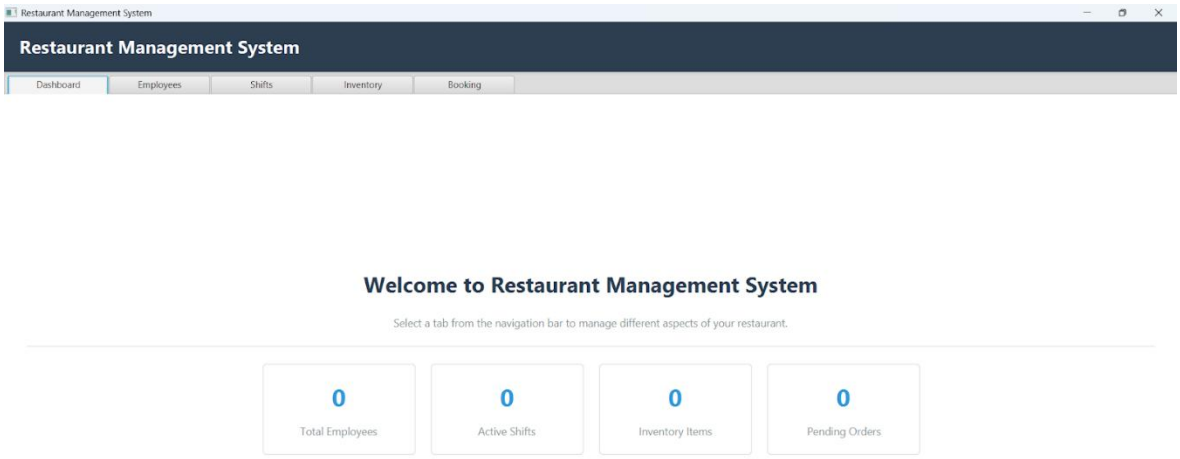
- **employeeController, shiftController, ...:** Lưu giữ các instance của Controller, có tác dụng giúp đảm bảo các Controller sống xuyên suốt vòng đời ứng dụng và có thể giao tiếp với nhau (nếu cần).
- **...Table (TableView<T>):** Giúp lưu tham chiếu đến các bảng dữ liệu chính, có tác dụng dùng để refresh giao diện từ MainApp hoặc chuyển đổi giữa các Tab.
- **currentWeekStart (LocalDate):** Có tác dụng giúp quản lý trạng thái ("State") của Lịch làm việc. Xác định xem người dùng đang xem tuần nào để render đúng dữ liệu cho tuần đó.

2.2 Thiết kế giao diện người dùng:

Về tổng quan thiết kế, ứng dụng được thiết kế dựa theo ứng dụng Desktop của nền tảng JavaFX, có thiết kế phẳng, bố cục gọn gàng. Cách điều hướng trên ứng dụng: Sử dụng các thanh Tab Navigation nằm ngang ở góc trái trên, bao gồm 5 tab chính: Dashboard, Employees, Shifts, Inventory và Booking.

Chi tiết về từng màn hình chức năng:

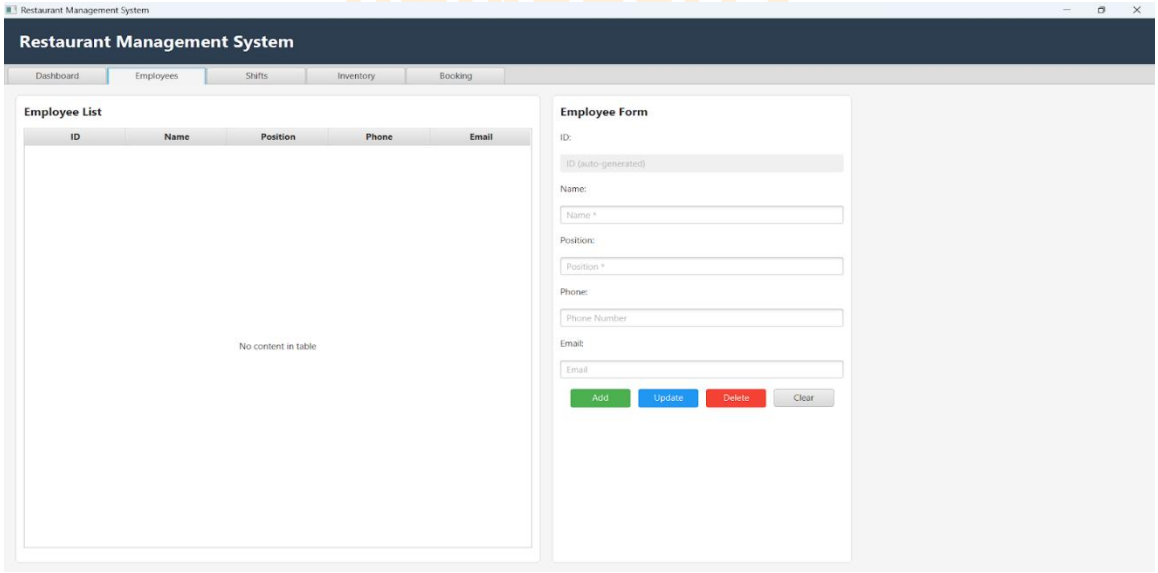
1. **Dashboard Tab:**



Hình 2-2. Dashboard Tab

Bố cục: hiển thị thông tin tổng quan của 4 tab employees shifts, Inventory và Booking bằng các chỉ số tổng nhân viên, tổng các ca làm việc hiện tại, hàng tồn kho và các đơn hàng chờ giải quyết.

2. Employees Tab

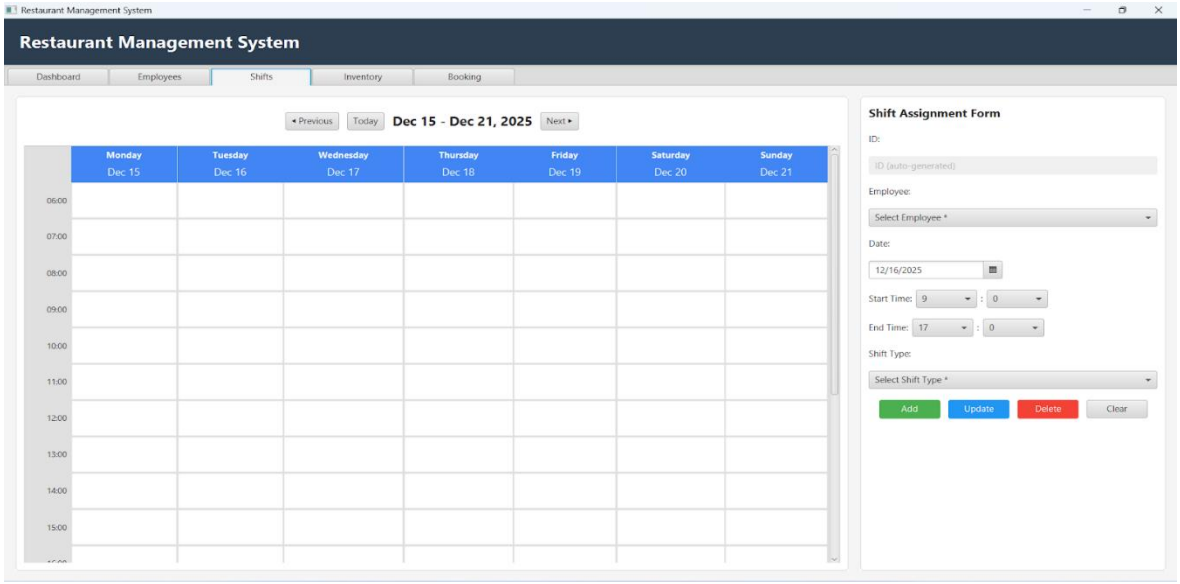


Hình 2-3. Employees Tab

– **Bố cục hiển thị:** Danh sách nằm ở bên trái và chi tiết dữ liệu nhập liệu ở bên phải.

- **Ở bên trái sẽ có bảng dữ liệu (TableView) với các cột:** ID, name, Position, Phone, Email.
- **Ở bên phải** có Form nhập liệu xếp dọc. Tại đây, nhà quản lý nhập tên, số điện thoại và email của từng nhân viên với số id riêng. Trường ID bị khoá vì máy sẽ tự động tạo ra một ID với mỗi đối tượng khác nhau. Ngoài ra có các nút chức năng như Add, Update, Delete, Clear giúp người dùng nhập nhân viên mới, chỉnh sửa thông tin nhân viên hoặc xoá đi tên nhân viên.

3. Shifts Tab



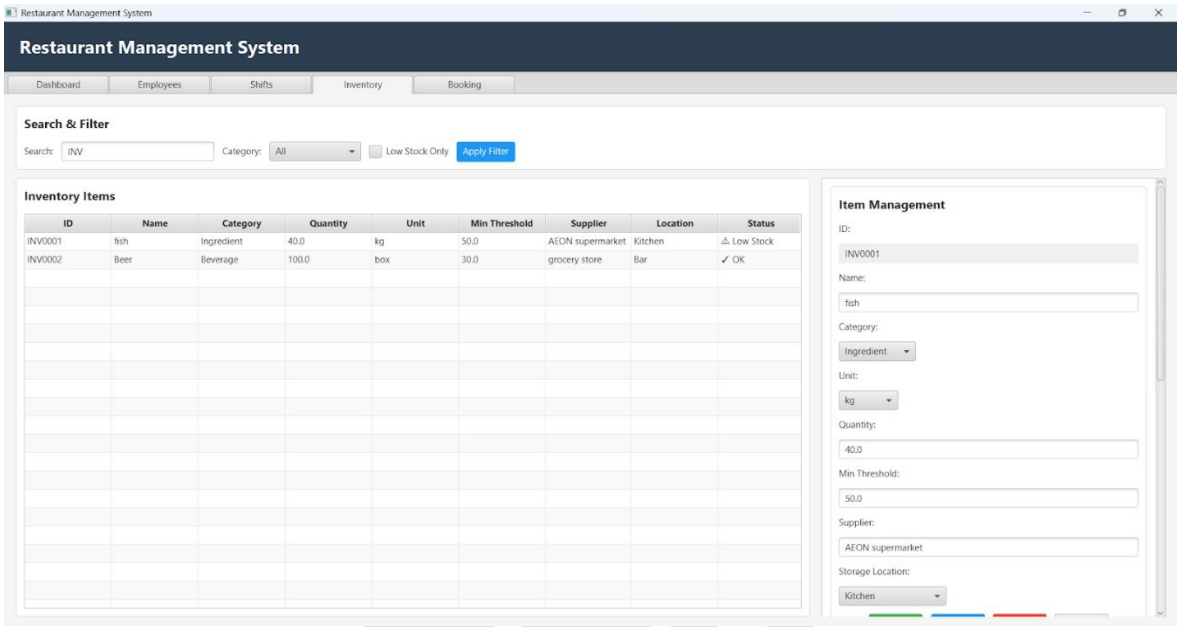
Hình 2-4. Shifts Tab

– **Bố cục hiển thị:** Lịch làm việc tuần ở bên trái và khu nhập liệu ở bên phải.

- **Ở bên trái sẽ có các header điều hướng tuần:** Có các nút Previous, Today, Next và hiển thị khoảng thời gian hiện tại của máy chủ mà người dùng đang dùng. (Ví dụ: Dec 22 Dec 28, 2025)
  - **Lưới lịch:** Gồm các cột là các ngày trong tuần từ thứ 2 chủ nhật, hàng là các khung giờ từ 6:00 đến 24:00.
  - **Hiển thị ca làm:** Khối màu xanh dương được hiển thị trực quan trên lưới, có ghi chú giờ rõ ràng để người dùng nắm bắt được lịch trình rõ ràng hơn.

- **Khu vực bên phải:** Form để điền thông tin ca làm của nhân viên bằng cách Dropdown chọn khung giờ, chọn ngày cũng như chọn giờ bắt đầu/kết thúc ca. Chúng tôi còn thêm cả lựa chọn ca làm việc là sáng, chiều hoặc tối ở nút shift type

4. Inventory Tab



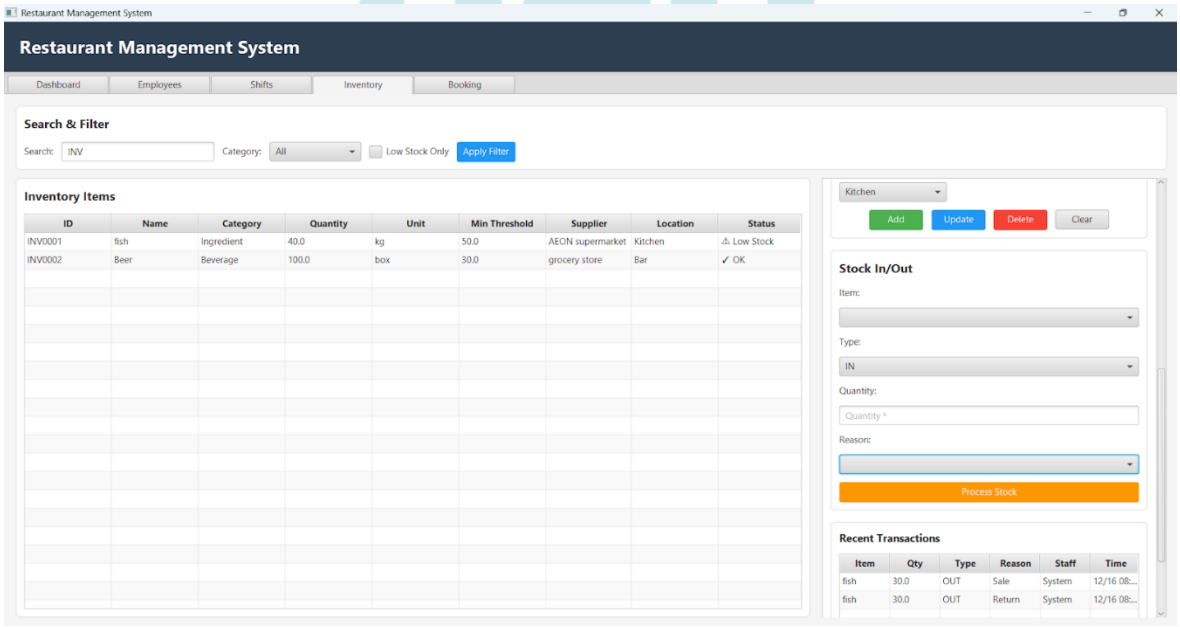
Hình 2-5. Inventory Tab

- **Giao diện của tab này có nhiều phân vùng chức năng:**
  - **Thanh công cụ trên (Search & Filter):** Đầu tiên là ô tìm kiếm để tìm kiếm hàng hóa dựa trên tên và ID món hàng. Tiếp theo Dropdown chọn danh mục (Category), Ch1eckbox "Low Stock Only" và nút "Apply Filter" màu xanh dương dùng để lọc hàng hóa theo điều kiện khác nhau
  - **Bảng dữ liệu (Inventory Items):** Hiển thị thông tin hàng hóa, nhà cung cấp, vị trí. Đầu tiên là cột Status (trạng thái) thể hiện bằng icon trực quan. Dấu tích (✓ OK) cho biết nhà hàng đáng không thiếu hàng, biển báo tam giác (⚠ Low Stock) cho biết hàng hóa dưới ngưỡng tối thiểu và cảnh báo cho nhà quản lý về sự thiếu hụt hàng tồn
- **Form thao tác (Phải):**
  - **Item Management:** Đây là form thêm/sửa thông tin hàng hóa. Đầu tiên là nhập tên hàng hóa mới. Nút category phân hàng hóa thành năm loại

là thức ăn (food), đồ uống (beverage), nguyên liệu nấu (ingredient), dụng cụ vệ sinh (cleaning), hàng hóa khác (other). Sau đó, quản lý sẽ nhập số lượng với đơn vị tùy chọn, mức dự trữ tối thiểu của hàng tồn kho và tên đơn vị cung cấp hàng hóa

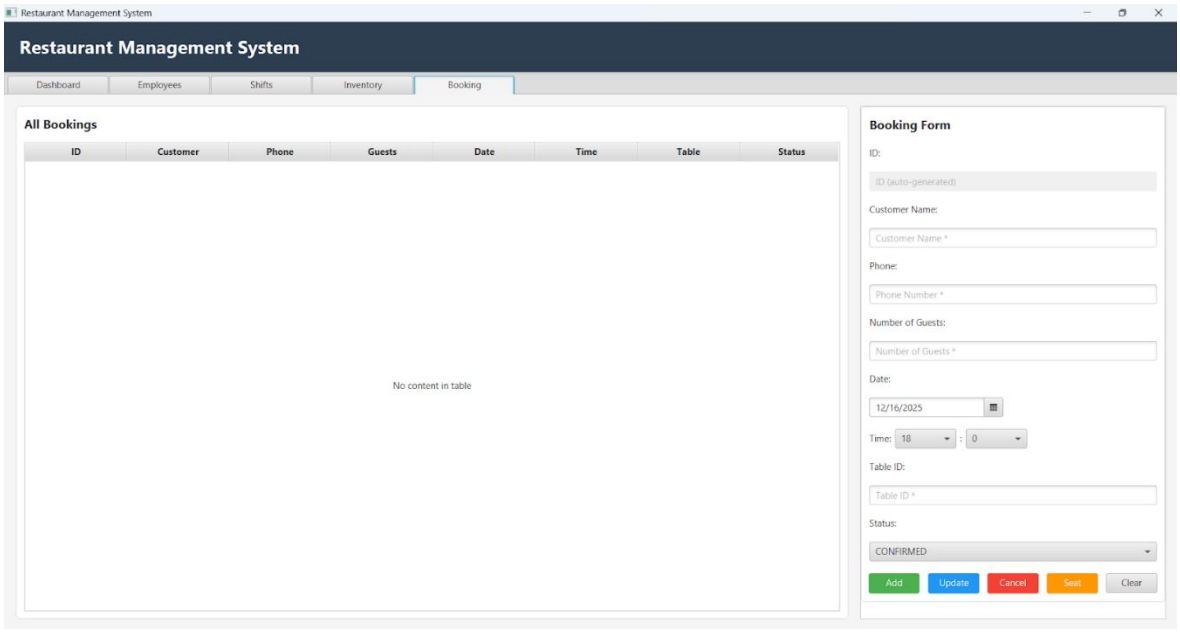
- **Stock In/Out:** Form xử lý nhập xuất kho với nút "Process Stock" với màu cam nổi bật. Tại mục này, nhà quản lý ghi nhận hàng hóa nhập kho (In) và xuất kho (Out) với các lý do gồm mua hàng (purchase), bán hàng (Sale), hao phí (Waste), điều chỉnh hàng tồn (Adjustment) và trả lại hàng (return).
- **Recent Transactions:** Bảng phụ nhỏ góc dưới cùng bên phải hiển thị lịch sử giao dịch gần nhất dựa trên thông tin của Stock In/Out

– **Thông báo:** Sử dụng Modal Popup (Hộp thoại) báo "Successfully" khi thao tác thành công và cảnh báo nếu thao tác không đúng



Hình 2-6. Modal Popup

5. Booking Tab



– **Bố cục:** Tương tự tab Employee (Bảng bên trái, Form nhập liệu bên phải).
  - **Bảng dữ liệu (All Bookings):** Hiển thị ID, Khách hàng, SĐT, Số khách, Ngày giờ, Bàn và Trạng thái.
  - **Form đặt bàn:**
    - Đầu tiên là tên khách hàng . Sau đó là số điện thoại của khách và số lượng người muốn đến nhà hàng. Sau đó, quản lý sẽ nhập thời gian phục vụ và ID bàn cho khách.
    - Dropdown chọn trạng thái của bàn. Nếu khách hàng đã gọi điện đặt rồi thì trạng thái là Confirmed. Nếu khách hàng đã đến nhà hàng thì trạng thái là Seat còn nếu khách hàng đã hủy bỏ đơn đặt thì là Canceled
    - Nút chức năng đặc thù: Có thêm nút màu cam "Seat" (xếp bàn) bên cạnh các nút CRUD tiêu chuẩn, dùng để chuyển trạng thái khách đã đến.

Chương 3 Hiện thực ứng dụng

3.1 Ngôn ngữ và công cụ hiện thực

Mục này trình bày chi tiết các ngôn ngữ lập trình, công nghệ và công cụ được sử dụng để hiện thực hệ thống quản lý nhà hàng. Khác với Chương 2 tập trung vào phân tích và thiết kế, mục 3.1 làm rõ cách các thành phần đã được thiết kế được chuyển hóa thành mã nguồn cụ thể trong môi trường triển khai thực tế.

Hệ thống được hiện thực dưới dạng một ứng dụng desktop viết bằng Java, với kiến trúc phân lớp rõ ràng nhằm tách biệt các thành phần trong hệ thống. Source code được tổ chức theo mô hình package chuẩn của Java, giúp tăng tính rõ ràng trong quản lý mã nguồn cũng như tạo điều kiện thuận lợi cho việc bảo trì và mở rộng về sau. Giao diện người dùng được xây dựng hoàn toàn bằng code Java, cho phép kiểm soát trực tiếp hành vi của từng thành phần giao diện. Dữ liệu của hệ thống được quản lý trong bộ nhớ thông qua các cấu trúc dữ liệu chuẩn của Java, phù hợp với mục tiêu demo và kiểm thử chức năng của đề tài.

3.1.1 Môi trường phát triển

Hệ thống được phát triển bằng ngôn ngữ lập trình Java, sử dụng Java Development Kit (JDK) phiên bản 17. Phiên bản này được xác định thông qua cấu hình của Maven Compiler Plugin, trong đó các thuộc tính maven.compiler.source và maven.compiler.target đều được thiết lập ở mức 17. Việc sử dụng Java 17 giúp hệ thống tận dụng các cải tiến về hiệu năng, độ ổn định và các tính năng ngôn ngữ hiện đại, đồng thời đảm bảo khả năng tương thích tốt với JavaFX phiên bản tương ứng.

Dự án sử dụng Apache Maven làm công cụ quản lý dự án và dependencies. File pom.xml đóng vai trò trung tâm trong việc định nghĩa cấu trúc dự án, quản lý thư viện, cấu hình biên dịch và hỗ trợ quá trình chạy ứng dụng. Các thông tin như groupId, artifactId và version được khai báo rõ ràng, giúp định danh dự án một cách nhất quán trong quá trình phát triển.

Trong cấu hình Maven, một số plugin quan trọng được sử dụng nhằm hỗ trợ quá trình hiện thực và chạy thử hệ thống. Maven Compiler Plugin đảm nhiệm việc biên dịch source code theo chuẩn Java 17. JavaFX Maven Plugin được cấu hình để

chạy ứng dụng JavaFX với class khởi động chính là com.restaurantmanagement.app.MainApp. Ngoài ra, Exec Maven Plugin được sử dụng như một phương thức bổ trợ để thực thi ứng dụng trong quá trình phát triển và kiểm thử. Sự kết hợp của các công cụ này giúp đơn giản hóa quá trình build, đảm bảo tính nhất quán giữa các môi trường phát triển khác nhau và giảm thiểu các vấn đề liên quan đến cấu hình thủ công.

3.1.2 Công nghệ giao diện người dùng

Giao diện người dùng của hệ thống được xây dựng dựa trên công nghệ JavaFX. Việc sử dụng JavaFX được xác định trực tiếp từ các import statement trong class khởi động MainApp cũng như trong các lớp controller. JavaFX phiên bản 17.0.2 được khai báo trong file pom.xml và được sử dụng xuyên suốt trong toàn bộ ứng dụng.

Toàn bộ giao diện của hệ thống được xây dựng hoàn toàn bằng code Java. Trong class MainApp, các thành phần giao diện như scene, layout và control được khởi tạo trực tiếp thông qua các constructor và phương thức của JavaFX. Cách tiếp cận này cho phép kiểm soát chi tiết cấu trúc giao diện, đồng thời giúp việc gắn kết giữa giao diện và logic xử lý trở nên trực tiếp và rõ ràng hơn.

Các thành phần giao diện trong hệ thống bao gồm các layout container như VBox, HBox, GridPane, StackPane và BorderPane, được sử dụng để tổ chức bố cục giao diện theo từng khu vực chức năng. Bên cạnh đó, các control như TableView, TextField, Button, ComboBox, DatePicker, Label và CheckBox được sử dụng để hỗ trợ việc nhập liệu và hiển thị dữ liệu. Việc điều hướng giữa các module chức năng của hệ thống được thực hiện thông qua TabPane và Tab, cho phép chia giao diện thành các khu vực độc lập tương ứng với từng nhóm chức năng. Các thành phần hỗ trợ hiển thị như ScrollPane và Separator được sử dụng để cải thiện khả năng trình bày và trải nghiệm người dùng.

Giao diện người dùng tương tác với hệ thống thông qua các lớp controller, mỗi controller phụ trách một module chức năng cụ thể. Các controller tiếp nhận sự kiện từ giao diện, quản lý dữ liệu hiển thị thông qua ObservableList và gọi các service tương ứng để xử lý logic nghiệp vụ. Cách tổ chức này giúp đảm bảo sự tách biệt rõ ràng giữa giao diện và xử lý nghiệp vụ.

3.1.3 Quản lý dữ liệu

Hệ thống áp dụng phương pháp quản lý dữ liệu trong bộ nhớ (in-memory storage), trong đó toàn bộ dữ liệu được lưu trữ trong RAM và không được ghi xuống thiết bị lưu trữ lâu dài. Điều này được thể hiện rõ thông qua việc sử dụng các lớp repository với tiền tố InMemory, chẳng hạn như:

- InMemoryEmployeeRepository
- InMemoryBookingRepository
- InMemoryInventoryRepository
- InMemoryShiftRepository
- InMemoryInventoryTransactionRepository

Trong mỗi repository, dữ liệu được lưu trữ bằng các cấu trúc dữ liệu chuẩn của Java, chủ yếu là ArrayList. Mỗi repository duy trì một danh sách các entity tương ứng và cung cấp các phương thức để thực hiện các thao tác thêm mới, cập nhật, truy vấn và xóa dữ liệu. Việc quản lý mã định danh cho các entity được thực hiện thông qua một biến đếm nội bộ, cho phép hệ thống tự động sinh ID theo định dạng có tiền tố, giúp phân biệt rõ từng loại dữ liệu.

Các thao tác truy vấn và xử lý dữ liệu được triển khai bằng cách kết hợp các phương thức của collection framework và Stream API của Java. Cách tiếp cận này giúp mã nguồn ngắn gọn, rõ ràng và dễ đọc, đồng thời hỗ trợ linh hoạt các yêu cầu tìm kiếm theo nhiều tiêu chí khác nhau.

Việc lựa chọn mô hình lưu trữ in-memory phù hợp với mục tiêu hiện thực và demo hệ thống, cho phép tập trung vào việc kiểm thử logic nghiệp vụ mà không phụ thuộc vào cơ sở dữ liệu bên ngoài. Tuy nhiên, do dữ liệu không được lưu trữ lâu dài, toàn bộ thông tin sẽ bị mất khi ứng dụng kết thúc.

3.1.4 Thư viện và API sử dụng

Các thư viện và API được sử dụng trong hệ thống được xác định trực tiếp từ file pom.xml và các import statement trong source code. Nhóm thư viện quan trọng nhất là các module JavaFX, bao gồm javafx-base, javafx-controls và javafx-graphics,

với phiên bản 17.0.2. Các module này cung cấp nền tảng để xây dựng giao diện người dùng, quản lý các thành phần giao diện, xử lý sự kiện và hiển thị nội dung trên màn hình. Module javafx-fxml cũng được khai báo trong Maven, tuy nhiên không được sử dụng trong quá trình hiện thực giao diện.

Bên cạnh JavaFX, hệ thống sử dụng rộng rãi các API từ Java Standard Library. Package java.util được sử dụng để quản lý các cấu trúc dữ liệu và thực hiện các thao tác xử lý collection thông qua Stream API. Package java.time cung cấp các class xử lý ngày giờ, phục vụ cho các chức năng liên quan đến đặt bàn, ca làm việc và quản lý thời gian. Ngoài ra, package javafx.collections được sử dụng để quản lý các danh sách dữ liệu có thể quan sát được, cho phép giao diện tự động cập nhật khi dữ liệu thay đổi.

Toàn bộ các thư viện này được quản lý tập trung thông qua Maven, giúp đảm bảo tính nhất quán về phiên bản và đơn giản hóa quá trình nâng cấp cũng như bảo trì hệ thống.

3.1.5 Nguyên tắc hiện thực

Quá trình hiện thực hệ thống tuân thủ các nguyên tắc thiết kế phần mềm và lập trình hướng đối tượng nhằm đảm bảo tính rõ ràng, dễ bảo trì và khả năng mở rộng. Trước hết, hệ thống được tổ chức theo kiến trúc phân lớp với các lớp Model, Repository, Service và Controller, trong đó mỗi lớp đảm nhận một vai trò và trách nhiệm riêng biệt. Cách tổ chức này giúp tách biệt rõ ràng giữa dữ liệu, logic nghiệp vụ và giao diện người dùng.

Bên cạnh đó, hệ thống tuân thủ nguyên tắc tách biệt mối quan tâm, khi mỗi thành phần chỉ tập trung vào một nhiệm vụ cụ thể và không can thiệp vào trách nhiệm của các thành phần khác. Các nguyên tắc lập trình hướng đối tượng như đóng gói và trừu tượng hóa được áp dụng xuyên suốt trong code, thể hiện qua việc sử dụng các field private, các phương thức truy cập công khai và việc che giấu chi tiết cài đặt của tầng lưu trữ dữ liệu.

Cấu trúc hiện tại cũng tạo điều kiện thuận lợi cho việc bảo trì và mở rộng hệ thống trong tương lai. Việc sử dụng repository pattern cho phép dễ dàng thay thế mô hình lưu trữ in-memory bằng cơ sở dữ liệu mà không ảnh hưởng đến các tầng service

và controller. Đồng thời, cách tổ chức code theo từng module chức năng giúp việc bổ sung hoặc mở rộng chức năng mới được thực hiện một cách độc lập và có kiểm soát, phù hợp với định hướng phát triển lâu dài của hệ thống.

3.2 Tầng Application

Tầng Application, được hiện thực thông qua lớp MainApp, đóng vai trò là **điểm khởi động (entry point)** và **trung tâm điều phối (orchestration center)** của toàn bộ hệ thống. Trong kiến trúc phân lớp đã được trình bày ở Chương 2, tầng này nằm ở vị trí cao nhất, chịu trách nhiệm khởi tạo vòng đời ứng dụng, thiết lập và điều phối các mối quan hệ phụ thuộc giữa các controller và service, đồng thời quản lý luồng tổng thể của giao diện người dùng.

3.2.1 Cấu trúc Class và khai báo Field

a) Khai báo Class và kế thừa:

```
public class MainApp extends Application {
```

Việc MainApp kế thừa từ class Application của JavaFX là bước thiết yếu xác định đây là điểm khởi đầu chuẩn cho ứng dụng desktop, hiện thực hóa đúng vai trò lớp khởi tạo trong thiết kế UML. Mối quan hệ này không chỉ cung cấp hạ tầng framework để quản lý luồng (thread) và quy trình khởi chạy, mà còn cho phép MainApp thừa hưởng toàn bộ các phương thức vòng đời ứng dụng—đặc biệt là start(Stage)—để kích hoạt và điều phối hệ thống ngay từ khi bắt đầu.

b) Khai báo Field Controller

```
private EmployeeController employeeController;
private ShiftController shiftController;
private InventoryController inventoryController;
private BookingController bookingController;
```

Bốn field instance này thiết lập các phụ thuộc controller thiết yếu cho MainApp, hiện thực hóa trực tiếp quan hệ phụ thuộc (uses) trong biểu đồ UML đồng thời đảm bảo an toàn kiểu tại thời điểm biên dịch. Bằng cách sử dụng phạm vi private, hệ thống duy trì tính đóng gói và ranh giới kiến trúc của tầng Application, ngăn chặn

truy cập trực tiếp từ bên ngoài. Đặc biệt, việc tồn tại xuyên suốt vòng đời ứng dụng cho phép MainApp giữ tham chiếu ổn định để điều phối linh hoạt các controller này qua nhiều ngữ cảnh xử lý khác nhau.

c) Khai báo Filed Tableview

```
private TableView<Employee> employeeTable;
private TableView<Shift> shiftTable;
private TableView<InventoryItem> inventoryTable;
private TableView<InventoryTransaction> transactionTable;
private TableView<Booking> bookingTable;
```

Việc khai báo năm đối tượng TableView với tham số generic cụ thể (như <Employee>, <Shift>) không chỉ định hình rõ ràng thành phần hiển thị dữ liệu cho từng thực thể, mà còn thiết lập cơ chế an toàn kiểu chặt chẽ và hỗ trợ PropertyValueFactory tự động ràng buộc thuộc tính model. Quan trọng hơn, bằng cách duy trì các bảng này dưới dạng biến instance thay vì cục bộ, hệ thống đảm bảo khả năng truy cập xuyên suốt, cho phép chia sẻ tham chiếu giữa các phương thức, event handler và Controller để thực hiện thao tác ràng buộc dữ liệu nhất quán trong toàn bộ vòng đời ứng dụng.

d) Khai báo Field Form Input

```
private TextField idField, nameField, positionField, phoneField,
emailField;
private TextField shiftIdField;
private ComboBox<String> employeeComboBox;
private DatePicker datePicker;
private ComboBox<Integer> startHourComboBox, startMinuteComboBox;
private ComboBox<Integer> endHourComboBox, endMinuteComboBox;
private ComboBox<String> shiftTypeComboBox;
```

Các dòng 48-54 thiết lập cấu trúc cho các thành phần nhập liệu thông qua việc áp dụng linh hoạt các pattern khai báo Java: gom nhóm các TextField cùng chức năng trên một dòng để tối ưu sự gọn gàng, đồng thời tách biệt các thành phần đặc thù để đảm bảo tính rõ ràng. Đặc biệt, việc sử dụng tham số generic (như ComboBox<String>) không chỉ định danh chính xác kiểu dữ liệu mà còn tăng cường độ an toàn kiểu khi truy xuất giá trị. Mục đích cốt lõi của các field này là tạo ra các tham chiếu UI bền vững; sau khi khởi tạo, chúng sẽ được đăng ký trực tiếp với

Controller, cho phép logic nghiệp vụ tương tác lập trình (đọc/ghi) với dữ liệu người dùng một cách chính xác và hiệu quả.

e) Khai báo Field Trạng thái ứng dụng

```
private LocalDate currentWeekStart;
```

Field currentWeekStart tại dòng 55 đóng vai trò then chốt trong việc duy trì trạng thái cấp ứng dụng cho giao diện lịch (Calendar View). Việc lựa chọn kiểu dữ liệu java.time.LocalDate là quyết định thiết kế tối ưu, bởi nó không chỉ mô tả chính xác khái niệm ngày (tách biệt khỏi thông tin thời gian) mà còn đảm bảo an toàn dữ liệu nhờ tính bất biến (immutability) và cung cấp bộ công cụ thao tác thời gian mạnh mẽ (như plusDays, minusWeeks). Mặc dù không được mô tả tường minh trong biểu đồ UML, field này là thành phần hiện thực thiết yếu, đóng vai trò như "bộ nhớ" giúp ứng dụng lưu giữ ngữ cảnh hiển thị xuyên suốt các chuỗi tương tác UI của người dùng.

3.2.2 Điểm khởi đầu của ứng dụng: Method start()

a) Chữ ký phương thức

```
    @Override
    public void start(Stage primaryStage) {
```

Annotation @Override tại dòng 57 không chỉ xác nhận việc hiện thực hóa phương thức trừu tượng start(Stage) từ lớp cơ sở Application mà còn đánh dấu đây là điểm nhập (entry point) chính thức của vòng đời ứng dụng, được JavaFX runtime tự động kích hoạt. Với chữ ký public void start(Stage primaryStage) tuân thủ chính xác thiết kế UML, phương thức này tiếp nhận đối tượng primaryStage—đại diện cho cửa sổ hiển thị chính—và đóng vai trò trung tâm điều phối toàn bộ chuỗi khởi tạo, chuyển đổi ứng dụng từ trạng thái tĩnh sang trạng thái hoạt động.

b) Khởi tạo Service và Controller

```
    EmployeeService sharedEmployeeService = new EmployeeService();
    employeeController = new EmployeeController(sharedEmployeeService);
    shiftController = new ShiftController(sharedEmployeeService);
```

Đoạn code từ dòng 59 đến 62 hiện thực hóa quyết định thiết kế then chốt về tính nhất quán dữ liệu thông qua mẫu thiết kế Dependency Injection. Bằng cách khởi

tạo một instance duy nhất sharedEmployeeService và tiêm (inject) nó vào cả EmployeeController và ShiftController, MainApp đóng vai trò kiểm soát phụ thuộc, đảm bảo hai phân hệ này cùng hoạt động trên một nguồn dữ liệu duy nhất ("single source of truth"). Cách tiếp cận này triệt tiêu nguy cơ phân mảnh dữ liệu—vốn xảy ra nếu mỗi controller tự khởi tạo service riêng lẻ—đồng thời đảm bảo mọi thao tác cập nhật nhân viên tại EmployeeController sẽ được phản ánh tức thời sang ShiftController do cùng chia sẻ tầng InMemoryEmployeeRepository bên dưới.

```
    inventoryController = new InventoryController();
    bookingController = new BookingController();
```

Tại dòng 63-64, việc khởi tạo InventoryController và BookingController mà không thông qua cơ chế Dependency Injection bên ngoài cho thấy các lớp này tự quản lý instance service nội bộ, phản ánh sự linh hoạt trong thiết kế kiến trúc dựa trên đặc thù nghiệp vụ. Trái ngược với sự ràng buộc dữ liệu chặt chẽ giữa EmployeeController và ShiftController (cần chia sẻ trạng thái của entity Employee), hai controller này vận hành trên các domain độc lập (Inventory và Booking), do đó việc không chia sẻ service giúp đơn giản hóa mô hình khởi tạo mà vẫn đảm bảo tính biệt lập cần thiết cho từng phân hệ.

c) Khởi tạo trạng thái ứng dụng

```
    // Initialize current week to start of current week
    currentWeekStart                                    =
LocalDate.now().with(java.time.DayOfWeek.MONDAY);
```

Đoạn code tại dòng 66-67 thiết lập mốc thời gian khởi điểm cho ứng dụng bằng cách sử dụng LocalDate.now() kết hợp với toán tử điều chỉnh .with(java.time.DayOfWeek.MONDAY). Logic này đảm bảo rằng bất kể người dùng khởi chạy ứng dụng vào thời điểm nào, trạng thái lịch (Calendar View) sẽ luôn tự động căn chỉnh và hiển thị tuần làm việc hiện tại bắt đầu từ thứ Hai—tuân thủ quy chuẩn hiển thị thời gian mặc định của hệ thống.

d) Cấu hình Primary Stage

```
    primaryStage.setTitle("Restaurant Management System");
    primaryStage.setWidth(1600);
    primaryStage.setHeight(900);
    primaryStage.setMinWidth(1200);
```

```
    primaryStage.setMinHeight(700);
```

Đoạn code từ dòng 69 đến 73 thiết lập các thuộc tính hiển thị cốt lõi cho cửa sổ chính, bắt đầu bằng việc định danh ứng dụng rõ ràng qua setTitle. Kích thước khởi tạo được cấu hình ở mức rộng rãi (1600x900 pixel) nhằm cung cấp không gian hiển thị tối ưu cho các thành phần UI phức tạp như bảng dữ liệu và lịch biểu. Đồng thời, các ràng buộc kích thước tối thiểu (1200x700) được áp dụng chặt chẽ để bảo vệ trải nghiệm người dùng, ngăn chặn tình trạng giao diện bị phá vỡ hoặc che khuất các chức năng quan trọng khi người dùng thu nhỏ cửa sổ quá mức.

e) Tạo UI và hiển thị giao diện người dùng

```
    VBox root = createMainLayout();
    Scene scene = new Scene(root);
    primaryStage.setScene(scene);
    primaryStage.show();
```

Đoạn code từ dòng 75 đến 78 hoàn tất chuỗi khởi tạo giao diện thông qua quy trình lắp ráp phân cấp: bắt đầu bằng việc gọi createMainLayout() để xây dựng container gốc (VBox), sau đó đóng gói nó vào đối tượng Scene (dòng 76) và thiết lập liên kết với cửa sổ chính qua primaryStage.setScene() (dòng 77). Lệnh primaryStage.show() ở dòng 78 là bước quyết định cuối cùng; nó không chỉ hiển thị giao diện trực quan mà còn chính thức chuyển giao quyền kiểm soát cho JavaFX Application Thread, đưa ứng dụng từ trạng thái khởi tạo sang trạng thái tương tác thực tế với người dùng.

3.2.3 Tạo Main Layout: Method createMainLayout()

a) Khai báo Method và Root Container

```
    private VBox createMainLayout() {
        VBox root = new VBox();
        root.setStyle("-fx-background-color: #f5f5f5;");
```

Method createMainLayout được thiết kế là một helper method nội bộ (private), chịu trách nhiệm khởi tạo và trả về một đối tượng VBox—container đóng vai trò là node gốc (root node) sắp xếp toàn bộ thành phần giao diện theo trục dọc. Tại dòng 86, việc áp dụng styling CSS nội tuyến (-fx-background-color: #f5f5f5;) không chỉ minh họa cú pháp đặc trưng của JavaFX (sử dụng tiền tố -fx-) mà còn thiết lập tông

màu nền trung tính, tạo sự nhất quán về thẩm mỹ cho toàn bộ ứng dụng ngay từ lớp layout cơ sở.

b) Tạo Title Bar và Tab Panel

```
    // Title Bar
    HBox titleBar = createTitleBar();

    // Tab Navigation
    TabPane tabPane = createTabPane();
    tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.UNA
VAILABLE);
```

Việc phân tách logic khởi tạo UI vào các phương thức chuyên biệt như createTitleBar() và createTabPane() không chỉ tối ưu hóa tổ chức code mà còn thiết lập khung điều hướng chính cho toàn bộ hệ thống. Trong đó, cấu hình tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.UNAVAILABLE) đóng vai trò then chốt trong thiết kế trải nghiệm người dùng (UX): bằng cách vô hiệu hóa hoàn toàn chức năng đóng tab, ứng dụng cưỡng chế một cấu trúc điều hướng bền vững, đảm bảo người dùng luôn duy trì quyền truy cập vào các module chức năng cốt lõi (Dashboard, Employees, v.v.) và ngăn chặn mọi sự gián đoạn luồng công việc do thao tác sai lệch.

c) Kết nối Layout

```
    root.getChildren().addAll(titleBar, tabPane);
    VBox.setVgrow(tabPane, Priority.ALWAYS);

    return root;
```

Quy trình xây dựng layout hoàn tất bằng việc lắp ráp titleBar và tabPane vào container gốc thông qua phương thức addAll, xác lập trật tự hiển thị xếp chồng theo chiều dọc đặc trưng của VBox. Tuy nhiên, yếu tố quyết định trải nghiệm người dùng nằm ở thiết lập VBox.setVgrow(tabPane, Priority.ALWAYS) tại dòng 96. Lời gọi phương thức tĩnh này áp dụng ràng buộc layout động, ép buộc tabPane phải mở rộng để chiếm dụng toàn bộ không gian chiều dọc còn thừa, trong khi thanh tiêu đề giữ nguyên kích thước cố định. Cơ chế này tối ưu hóa diện tích hiển thị cho các chức năng nghiệp vụ chính trước khi đối tượng root hoàn chỉnh được trả về để khởi tạo Scene.

3.2.4 Xây dựng Tab Panel: Method createTabPane()

a) Tạo và cấu hình Tab Panel

```
    private TabPane createTabPane() {
        TabPane tabPane = new TabPane();
        tabPane.setStyle("-fx-tab-min-width: 120px;");
```

Phương thức nội bộ này chịu trách nhiệm khởi tạo đối tượng TabPane – container đóng vai trò xương sống cho hệ thống điều hướng. Điểm nhấn trong thiết kế nằm ở dòng 122 với thiết lập CSS nội tuyến (-fx-tab-min-width: 120px;); đây không chỉ là tinh chỉnh thẩm mỹ mà là giải pháp UX thiết yếu, cưỡng chế độ rộng tối thiểu đồng nhất cho các tab nhằm duy trì tính dễ đọc và ngăn chặn tình trạng giao diện bị biến dạng khi người dùng thay đổi kích thước cửa sổ.

b) Tạo Tab

```
    // Dashboard Tab
    Tab dashboardTab = createDashboardTab();

    // Employee Management Tab
    Tab employeeTab = createEmployeeTab();

    // Shift Assignment Tab
    Tab shiftTab = createShiftTab();

    // Inventory Tab
    Tab inventoryTab = createInventoryTab();

    // Booking Management Tab
    Tab bookingTab = createBookingTab();

    tabPane.getTabs().addAll(dashboardTab,  employeeTab,
shiftTab, inventoryTab, bookingTab);
```

Đoạn code từ dòng 124 đến 139 thể hiện tư duy module hóa triệt để trong việc xây dựng giao diện. Thay vì dồn code vào một chỗ, mỗi tab chức năng được khởi tạo thông qua các phương thức factory chuyên biệt (như createDashboardTab, createEmployeeTab), giúp đóng gói hoàn toàn logic dựng UI (bảng, form) vào từng đơn vị độc lập. Quá trình lắp ráp cuối cùng được thực hiện tại dòng 139 thông qua phương thức addAll; tại đây, thứ tự truyền tham số vào danh sách ObservableList của

TabPane sẽ định đoạt trực tiếp trình tự hiển thị trực quan từ trái sang phải trên thanh điều hướng, đảm bảo cấu trúc giao diện nhất quán với thiết kế.

c) Tab Selection Listener

```
    // Add listener to refresh employee combo box when Shift tab is selected
    tabPane.getSelectionModel().selectedItemProperty().addListener((obs, oldTab, newTab) -> {
        if (newTab == shiftTab && shiftController != null) {
            shiftController.refreshEmployeeComboBox();
        }
    });
```

Đoạn code từ dòng 141 đến 148 thiết lập cơ chế đồng bộ hóa dữ liệu liên tab (inter-tab communication) thông qua việc gắn một ChangeListener vào thuộc tính selectedItemProperty của TabPane. Biểu thức lambda được kích hoạt mỗi khi người dùng chuyển tab, thực hiện kiểm tra logic tại dòng 143: xác nhận tab đích là "Shift" và đảm bảo shiftController đã sẵn sàng (lập trình phòng thủ). Hành động cốt lõi nằm ở dòng 144, nơi MainApp chủ động gọi shiftController.refreshEmployeeComboBox(). Cơ chế này đảm bảo tính nhất quán dữ liệu thời gian thực: bất kỳ nhân viên nào vừa được thêm ở tab Employee sẽ lập tức xuất hiện trong danh sách chọn của tab Shift mà không cần người dùng can thiệp thủ công, minh chứng cho vai trò điều phối trung tâm của lớp Application trước khi trả về đối tượng TabPane hoàn chỉnh.

3.2.5 Hiện thực Tab Employee: Method createEmployeeTab()

a) Tạo cấu trúc Tab

```
    private Tab createEmployeeTab() {
        Tab tab = new Tab("Employees");
        tab.setClosable(false);

        HBox employeeContent = new HBox(15);
        employeeContent.setPadding(new Insets(15));
        employeeContent.setStyle("-fx-background-color: #f5f5f5;");
```

Phương thức createEmployeeTab chịu trách nhiệm khởi tạo tab chức năng quản lý nhân viên, bắt đầu với việc định danh tab là "Employees". Điểm đáng chú ý

tại dòng 215 là lệnh tab.setClosable(false); dù chính sách đóng tab đã được thiết lập ở cấp TabPane, việc lặp lại cấu hình này tại cấp độ tab thể hiện tư duy lập trình phòng thủ (defensive programming), khẳng định rõ ràng ý định thiết kế là tab này phải luôn tồn tại. Về mặt bố cục, việc sử dụng HBox(15) kết hợp với setPadding(new Insets(15)) thiết lập một không gian làm việc thoáng đãng, phân tách rõ ràng giữa khu vực bảng dữ liệu và form nhập liệu, trong khi thiết lập màu nền đồng bộ (#f5f5f5) đảm bảo sự liền mạch về thị giác với tổng thể ứng dụng.

b) Tạo Selection và kết nối

```
    // Left side Employee Table
    VBox tableSection = createTableSection();
    tableSection.setPrefWidth(700);

    // Right side Form
    VBox formSection = createFormSection();
    formSection.setPrefWidth(400);

    employeeContent.getChildren().addAll(tableSection,
formSection);

    tab.setContent(employeeContent);
    return tab;
```

Quy trình lắp ráp nội dung tab tuân theo mô hình bố cục "chia đôi" (split-view), nơi các phân vùng chức năng được module hóa thành tableSection (hiển thị danh sách) và formSection (nhập liệu). Với chiều rộng cố định được thiết lập lần lượt là 700px và 400px, hai thành phần này được tích hợp vào container HBox (dòng 229), tạo nên layout ngang với bảng bên trái và form bên phải một cách tự nhiên. Bước cuối cùng gán container tổng hợp này làm nội dung chính (setContent) và trả về đối tượng Tab hoàn chỉnh đã khép lại quy trình khởi tạo giao diện quản lý nhân viên, sẵn sàng cho việc hiển thị và tương tác.

3.2.6 Phần Bảng Employee: Method createTableSection()

a) Tạo và cấu hình bảng nhân sự

```
    private VBox createTableSection() {
        VBox tableBox = new VBox(10);
        tableBox.setPadding(new Insets(10));
```

```
        tableBox.setStyle("-fx-background-color: white; -fx-
border-color: #ddd; -fx-border-radius: 5;");

        Label tableLabel = new Label("Employee List");
        tableLabel.setStyle("-fx-font-size: 16px; -fx-font-
weight: bold;");

        employeeTable = new TableView<>();
        employeeTable.setColumnResizePolicy(TableView.CONSTR
AINED_RESIZE_POLICY);
        employeeTable.setPrefHeight(600);
```

Phương thức này khởi tạo khu vực hiển thị dữ liệu dưới dạng một VBox được container hóa kỹ lưỡng. Từ dòng 551 đến 557, giao diện được định hình theo phong cách "card" (thẻ) thông qua CSS—với nền trắng, viền bo tròn và tiêu đề nổi bật—tạo nên sự phân cấp thị giác rõ ràng. Trọng tâm logic nằm ở các dòng 559-561, nơi employeeTable được khởi tạo và gán cho biến instance để quản lý trạng thái; đồng thời, việc áp dụng CONSTRAINED_RESIZE_POLICY đóng vai trò tối ưu hóa trải nghiệm người dùng, buộc các cột tự động giãn cách để lấp đầy chiều rộng bảng, loại bỏ thanh cuộn ngang và duy trì cấu trúc hiển thị cân đối với chiều cao cố định 600px.

```
        // Table columns
        TableColumn<Employee, String> idColumn = new TableColumn<>("ID");
        idColumn.setCellValueFactory(new PropertyValueFactory<>("id"));
        idColumn.setPrefWidth(80);

        TableColumn<Employee, String> nameColumn = new TableColumn<>("Name");
        nameColumn.setCellValueFactory(new PropertyValueFactory<>("name"));
        nameColumn.setPrefWidth(150);
```

Đoạn code từ dòng 563 đến 582 thiết lập "xương sống" hiển thị dữ liệu cho bảng thông qua cơ chế **Data Binding**. Điểm mấu chốt nằm ở PropertyValueFactory—lớp này đóng vai trò cầu nối, sử dụng Java Reflection để tự động ánh xạ thuộc tính "id" (chuỗi string) tới phương thức getter tương ứng getId() trong lớp Employee. Điều này giúp giảm thiểu code kết nối thủ công nhưng cũng đặt ra yêu cầu nghiêm ngặt: tên thuộc tính và tên phương thức getter phải tuân thủ chính xác quy chuẩn JavaBean (ví dụ: nếu phương thức là getEmployeeId thay vì getId, mapping sẽ thất bại).

Bên cạnh đó, việc thiết lập setPrefWidth() ở đây mang ý nghĩa đặc biệt khi kết hợp với CONSTRAINED_RESIZE_POLICY (đã cấu hình ở dòng 560). Vì bảng sẽ tự động co giãn, các giá trị pixel này (80, 150, v.v.) không còn là kích thước cố định

tuyệt đối, mà hoạt động như các **tỷ lệ trọng số (relative weights)**. Nó đảm bảo rằng cột Name (150px) sẽ luôn chiếm không gian rộng gần gấp đôi cột ID (80px), duy trì sự cân đối thị giác bất kể kích thước cửa sổ ứng dụng thay đổi ra sao.

b) Ràng buộc Controller

```
    // Bind table to controller
    employeeController.setTableView(employeeTable);
    employeeTable.setItems(employeeController.getEmployeeList());
```

Đoạn mã từ dòng 586 đến 588 đóng vai trò then chốt trong việc thiết lập kiến trúc Model-View-Controller (MVC), cụ thể là việc tích hợp giữa lớp giao diện (View) và lớp điều khiển (Controller). Thông qua phương thức employeeController.setTableView(employeeTable), ứng dụng thực hiện một thao tác "tiêm phụ thuộc" (dependency injection) theo kiểu Setter. Hành động này trao quyền kiểm soát trực tiếp thành phần UI employeeTable cho EmployeeController. Nhờ việc nắm giữ tham chiếu này, Controller không chỉ có khả năng quản lý trạng thái hiển thị mà còn có thể truy cập sâu vào SelectionModel của bảng để xử lý các tương tác phức tạp từ người dùng, chẳng hạn như bắt sự kiện chọn hàng hoặc điều phối focus, đảm bảo logic nghiệp vụ được thực thi chính xác trên thành phần giao diện.

employeeTable.setItems(employeeController.getEmployeeList()) thiết lập cơ chế "liên kết dữ liệu phản ứng" (Reactive Data Binding) dựa trên mẫu thiết kế Observer. Thay vì sao chép dữ liệu tĩnh, phương thức này liên kết TableView với ObservableList được quản lý bởi Controller. Cơ chế này tạo ra một luồng dữ liệu một chiều tự động hóa: TableView trở thành một "quan sát viên" (Observer) của danh sách dữ liệu. Do đó, mọi thao tác thêm, xóa hoặc sửa đổi phần tử trong danh sách tại Controller sẽ lập tức kích hoạt sự kiện cập nhật giao diện mà không cần can thiệp lập trình thủ công để vẽ lại bảng, đảm bảo tính đồng bộ hóa thời gian thực và sự tách biệt rõ ràng giữa logic dữ liệu và logic hiển thị.

c) Row Selection Event Handling

```
    // Handle row selection
    employeeTable.getSelectionModel().selectedItemProperty().addListener(
            (obs, oldSelection, newSelection) -> {
                if (newSelection != null) {
                    employeeController.handleTableSelection();
```

```
                }
            }
        );
```

Đoạn mã từ dòng 590 đến 594 thiết lập cơ chế tương tác người dùng thông qua mô hình lập trình hướng sự kiện (Event-Driven Programming). Bằng cách gắn một ChangeListener vào selectedItemProperty() của SelectionModel, ứng dụng không chỉ lắng nghe các tương tác chuột đơn thuần mà theo dõi sự thay đổi trạng thái chọn của dữ liệu thực tế. Biểu thức lambda tại dòng 592-593 đóng vai trò bộ lọc logic: điều kiện kiểm tra if (newSelection != null) là chốt chặn quan trọng nhằm loại bỏ các trường hợp bỏ chọn (deselection) hoặc trạng thái rỗng, đảm bảo rằng logic nghiệp vụ chỉ được kích hoạt khi có một đối tượng dữ liệu hợp lệ được xác định.

Sự phân tách trách nhiệm (Separation of Concerns) được thể hiện rõ nét tại dòng 594 với lệnh employeeController.handleTableSelection(). Tại đây, lớp MainApp chỉ đóng vai trò "ngòi nổ" phát hiện sự kiện, trong khi toàn bộ logic xử lý dữ liệu—như trích xuất thông tin nhân viên và điền vào form—được ủy quyền hoàn toàn cho Controller. Cuối cùng, dòng 599 hoàn tất quy trình khởi tạo giao diện bằng cách đóng gói tiêu đề và bảng dữ liệu vào container tableBox thông qua phương thức addAll, trả về một thành phần giao diện hoàn chỉnh và sẵn sàng tích hợp vào layout chính.

3.2.7 Phần Form Employee: Method createFormSection()

a) Tạo container cho form và fields

```
    private VBox createFormSection() {
        VBox formBox = new VBox(15);
        formBox.setPadding(new Insets(10));
        formBox.setStyle("-fx-background-color: white; -fx-
border-color: #ddd; -fx-border-radius: 5;");

        Label formLabel = new Label("Employee Form");
        formLabel.setStyle("-fx-font-size: 16px; -fx-font-
weight: bold;");

        // Form fields
        idField = new TextField();
        idField.setPromptText("ID (auto-generated)");
        idField.setEditable(false);
        idField.setStyle("-fx-background-color: #f0f0f0;");
```

Phương thức createFormSection khởi tạo khu vực nhập liệu thông qua một container VBox (dòng 606-609), được thiết kế để tối ưu hóa trải nghiệm thị giác và thao tác. Với thiết lập khoảng cách giãn dòng (spacing) 15 pixel và padding bao quanh, layout đảm bảo các trường nhập liệu không bị dồn nén, tạo cảm giác thoáng đãng. Đáng chú ý, việc tái sử dụng toàn bộ khối CSS "card-style" (nền trắng, viền bo tròn) tương tự như phần hiển thị bảng (createTableSection) giúp duy trì sự đồng bộ chặt chẽ về ngôn ngữ thiết kế (Design Language) trên toàn bộ giao diện ứng dụng.

Về mặt logic tương tác (UX/UI logic), đoạn mã từ dòng 615 đến 618 thể hiện sự xử lý tinh tế đối với trường dữ liệu định danh (idField). Khác với các trường thông tin thông thường, ID được cấu hình ở trạng thái "bất biến" đối với người dùng cuối:

- **Logic bảo vệ:** Lệnh setEditable(false) ngăn chặn mọi nỗ lực chỉnh sửa thủ công, đảm bảo tính toàn vẹn của dữ liệu hệ thống (ID do repository tự sinh).
- **Tín hiệu trực quan:** Việc thay đổi màu nền sang xám (#f0f0f0) kết hợp với promptText giải thích rõ ràng ("auto-generated") đóng vai trò là một chỉ dẫn giao diện (affordance), giúp người dùng nhận thức ngay lập tức đây là trường chỉ đọc (read-only).

Ngược lại, các trường dữ liệu còn lại (nameField, positionField, v.v.) từ dòng 620 đến 630 được khởi tạo theo khuôn mẫu chuẩn, duy trì trạng thái tương tác đầy đủ để phục vụ chức năng nhập liệu và cập nhật thông tin người dùng.

b) Tạo nút và Event Handler

```
    // Buttons
    HBox buttonBox = new HBox(10);
    buttonBox.setAlignment(Pos.CENTER);

    Button addButton = new Button("Add");
    addButton.setPrefWidth(80);
    addButton.setStyle("-fx-background-color: #4CAF50; -fx-
text-fill: white;");
    addButton.setOnAction(e -> {
        employeeController.handleAdd();
        shiftController.refreshEmployeeComboBox();
    });
```

Đoạn mã bắt đầu bằng việc thiết lập một thanh công cụ điều khiển (Control Bar) thông qua container HBox tại dòng 633-634. Việc sử dụng setAlignment(Pos.CENTER) và khoảng cách giãn dòng 10 pixel không chỉ tạo nên bố cục cân đối mà còn định hướng sự tập trung của người dùng vào nhóm chức năng chính. Đối với nút "Add", thiết kế giao diện tuân thủ chặt chẽ các nguyên tắc về **tín hiệu thị giác (Visual Affordance)** trong UI/UX: màu nền xanh lá (#4CAF50) được áp dụng (dòng 638) để biểu thị ngữ nghĩa của một hành động tích cực (tạo mới/thêm vào), trong khi chiều rộng cố định (setPrefWidth) đảm bảo tính đồng nhất về hình học cho toàn bộ bộ nút.

Về mặt kiến trúc phần mềm, cơ chế xử lý sự kiện tại dòng 639-641 minh họa rõ nét vai trò của MainApp trong mô hình MVC:

- **Sự ủy quyền (Delegation):** Thông qua biểu thức Lambda trong setOnAction, lớp View không tự xử lý logic nghiệp vụ mà ủy quyền hoàn toàn cho Controller (employeeController.handleAdd()). Điều này đảm bảo nguyên tắc "Separation of Concerns" (Phân tách mối quan tâm), giữ cho lớp giao diện tinh gọn.
- **Điều phối liên module (Inter-module Orchestration):** Điểm đặc biệt quan trọng nằm ở dòng 641: shiftController.refreshEmployeeComboBox(). Đây là minh chứng cho vai trò "nhạc trưởng" của lớp MainApp. Vì module "Employees" và module "Shifts" hoạt động độc lập nhưng chia sẻ dữ liệu chung (nhân viên), hành động thêm nhân viên mới đòi hỏi sự cập nhật tức thời trên danh sách phân ca. Việc gọi hàm refresh trực tiếp đảm bảo tính **nhất quán dữ liệu (Data Consistency)** trên toàn hệ thống ngay thời gian thực, ngăn chặn tình trạng người dùng không tìm thấy nhân viên vừa tạo khi chuyển sang tab phân ca.

Mô hình thiết kế này—bao gồm định dạng ngữ nghĩa và logic ủy quyền—được tái áp dụng nhất quán cho các nút Update, Delete và Clear (dòng 644-669), chỉ thay đổi về màu sắc (ví dụ: màu đỏ cho hành động nguy hiểm như Delete) và phương thức đích, tạo nên một trải nghiệm người dùng dễ đoán và an toàn.

c) Tạo Form Field

```
    // Set form fields in controller
    employeeController.setFormFields(idField,
nameField, positionField, phoneField, emailField);
```

Đoạn mã từ dòng 673 đến 689 đánh dấu bước hoàn thiện cốt lõi trong việc xây dựng form nhập liệu bằng cách thiết lập liên kết trực tiếp giữa các thành phần giao diện (UI) và logic điều khiển. Tại dòng 674, thông qua phương thức employeeController.setFormFields(...), ứng dụng thực hiện kỹ thuật "tiêm phụ thuộc" (dependency injection) ở cấp độ View, truyền tham chiếu của toàn bộ các TextField vào EmployeeController. Cơ chế này mở ra kênh giao tiếp hai chiều thiết yếu: Controller không chỉ có khả năng "đọc" dữ liệu đầu vào từ người dùng để thực hiện các thao tác thêm/sửa, mà còn có thể "ghi" dữ liệu ngược lại vào các trường form để hiển thị chi tiết khi một bản ghi được chọn, đảm bảo tính đồng bộ hóa chặt chẽ giữa tương tác người dùng và xử lý nghiệp vụ.

Sau khi thiết lập logic liên kết, quy trình lắp ráp giao diện (Layout Composition) diễn ra từ dòng 676 đến 689. Các thành phần như nhãn tiêu đề, trường nhập liệu và nhóm nút chức năng được thêm tuần tự vào container VBox gốc thông qua phương thức addAll. Thứ tự thêm vào này quyết định trực tiếp cấu trúc hiển thị phân cấp từ trên xuống dưới, tạo nên dòng chảy thị giác logic cho người dùng. Cuối cùng, phương thức trả về đối tượng formBox hoàn chỉnh, đóng gói toàn bộ phân hệ nhập liệu thành một khối giao diện độc lập, sẵn sàng để tích hợp vào layout chính của tab.

3.2.8 Phần Bảng Shift: Method createShiftTableSection()

Method này tạo phần bảng hiển thị danh sách ca làm việc cho tab Shift. Mặc dù tab Shift chủ yếu sử dụng calendar view, bảng này vẫn được cung cấp để hiển thị dữ liệu dạng bảng.

a) Tạo container và bảng

```
    private VBox createShiftTableSection() {
        VBox tableBox = new VBox(10);
        tableBox.setPadding(new Insets(10));
        tableBox.setStyle("-fx-background-color: white; -fx-
border-color: #ddd; -fx-border-radius: 5;");
```

```
        Label tableLabel = new Label("Shift List");
        tableLabel.setStyle("-fx-font-size: 16px; -fx-font-
weight: bold;");

        shiftTable = new TableView<>();
        shiftTable.setColumnResizePolicy(TableView.CONSTRAIN
ED_RESIZE_POLICY);
        shiftTable.setPrefHeight(600);
```

Phương thức createShiftTableSection (dòng 697-707) chịu trách nhiệm xây dựng phân hệ hiển thị dữ liệu lịch trình, tuân thủ chặt chẽ nguyên tắc **Tính nhất quán trong Thiết kế (Design Consistency)**. Cấu trúc mã nguồn tái sử dụng hoàn toàn khuôn mẫu khởi tạo container của module Nhân viên—từ việc định nghĩa VBox (dòng 698-700) đến các thiết lập CSS cho đường viền và tiêu đề (dòng 702-703). Việc duy trì ngôn ngữ thiết kế đồng bộ này không chỉ giúp giảm tải nhận thức cho người dùng khi chuyển đổi giữa các tab mà còn đơn giản hóa quy trình bảo trì mã nguồn thông qua sự lặp lại có chủ đích của các mẫu UI (UI Patterns).

Tại trung tâm của logic hiển thị, đối tượng TableView được khởi tạo và gán trực tiếp vào biến instance shiftTable (dòng 705). Đây là một quyết định kiến trúc quan trọng, mở rộng phạm vi truy cập của bảng ra ngoài phương thức cục bộ, cho phép các thành phần khác—đặc biệt là giao diện Lịch (Calendar View)—tương tác trực tiếp để đồng bộ hóa dữ liệu hoặc điền form khi người dùng chọn một ca làm việc cụ thể. Bên cạnh đó, các cấu hình hiển thị (dòng 706-707) với CONSTRAINED_RESIZE_POLICY và chiều cao cố định 600px đóng vai trò tối ưu hóa trải nghiệm người dùng, đảm bảo các cột tự động dàn trải hợp lý để lấp đầy không gian mà không xuất hiện thanh cuộn ngang, đồng thời cung cấp viewport đủ lớn cho lượng dữ liệu phức tạp.

b) Tạo Cột với Cell Value Factory tùy chỉnh

```
        TableColumn<Shift, String> idColumn = new TableColumn<>("ID");
        idColumn.setCellValueFactory(new PropertyValueFactory<>("id"));
        idColumn.setPrefWidth(80);

        TableColumn<Shift, String> employeeColumn = new TableColumn<>("Employee");
        employeeColumn.setCellValueFactory(new
PropertyValueFactory<>("employeeName"));
        employeeColumn.setPrefWidth(150);

        TableColumn<Shift, String> dateColumn = new TableColumn<>("Date");
        dateColumn.setCellValueFactory(cellData -> {
```

```
            LocalDate date = cellData.getValue().getDate();
            return new javafx.beans.property.SimpleStringProperty(
                date != null ? date.format(DateTimeFormatter.ofPattern("yyyy-MM-
dd")) : ""
            );
        });
        dateColumn.setPrefWidth(120);

        TableColumn<Shift, String> startTimeColumn = new TableColumn<>("Start");
        startTimeColumn.setCellValueFactory(cellData -> {
            var time = cellData.getValue().getStartTime();
            return new javafx.beans.property.SimpleStringProperty(
                time != null ? String.format("%02d:%02d", time.getHour(),
time.getMinute()) : ""
            );
        });
        startTimeColumn.setPrefWidth(80);

        TableColumn<Shift, String> endTimeColumn = new TableColumn<>("End");
        endTimeColumn.setCellValueFactory(cellData -> {
            var time = cellData.getValue().getEndTime();
            return new javafx.beans.property.SimpleStringProperty(
                time != null ? String.format("%02d:%02d", time.getHour(),
time.getMinute()) : ""
            );
        });
        endTimeColumn.setPrefWidth(80);

        TableColumn<Shift, String> typeColumn = new TableColumn<>("Type");
        typeColumn.setCellValueFactory(new PropertyValueFactory<>("shiftType"));
        typeColumn.setPrefWidth(100);
```

Đoạn mã thiết lập cột cho bảng Shift (dòng 710-747) thể hiện sự kết hợp giữa hai phương pháp liên kết dữ liệu (Data Binding) khác nhau để tối ưu hóa hiển thị. Đối với các trường dữ liệu nguyên thủy hoặc chuỗi đơn giản như ID và Type, phương thức PropertyValueFactory được sử dụng (dòng 710 và 745) để ánh xạ trực tiếp theo quy ước JavaBean. Đáng chú ý ở dòng 714, thuộc tính employeeName được liên kết như một **thuộc tính được tính toán (computed property)**. Điều này cho thấy tầng Model đã được thiết kế để hỗ trợ View bằng cách "làm phẳng" quan hệ đối tượng, giúp bảng hiển thị tên nhân viên trực tiếp mà không cần logic phức tạp tại tầng giao diện để truy xuất từ đối tượng Employee lồng nhau.

Tuy nhiên, điểm sáng kỹ thuật nằm ở việc xử lý các cột thời gian (Date, Start Time, End Time) từ dòng 718 đến 743. Thay vì dựa vào PropertyValueFactory mặc định (vốn chỉ gọi toString() một cách cứng nhắc), mã nguồn sử dụng **Lambda Expressions** để tùy biến CellValueFactory. Kỹ thuật này mang lại hai lợi ích cốt lõi:

– **Định dạng linh hoạt:** Cho phép tích hợp trực tiếp DateTimeFormatter để chuyển đổi đối tượng LocalDate/LocalTime thành chuỗi hiển thị thân thiện (ví dụ: "yyyy-MM-dd" hay "HH:mm") ngay tại thời điểm render.
– **An toàn dữ liệu (Null Safety):** Logic bên trong lambda chủ động kiểm tra null và trả về giá trị mặc định an toàn, ngăn chặn lỗi NullPointerException khi dữ liệu lịch trình chưa đầy đủ. Cách tiếp cận này chứng minh khả năng xử lý tinh tế của JavaFX đối với các kiểu dữ liệu phức tạp, đảm bảo dữ liệu thô từ Model được chuyển hóa thành thông tin hữu ích trên View.

c) Ràng buộc Controller và Xử lý Sự kiện

```
// Bind table to controller
shiftController.setTableView(shiftTable);
shiftTable.setItems(shiftController.getShiftList());

// Handle row selection
shiftTable.getSelectionModel().selectedItemProperty().addListener(
        (obs, oldSelection, newSelection) -> {
            if (newSelection != null) {
                shiftController.handleTableSelection();
            }
        }
);
```

Đoạn mã từ dòng 752 đến 762 tái khẳng định sự tuân thủ nghiêm ngặt mô hình MVC được áp dụng xuyên suốt ứng dụng. Tại dòng 752-753, cơ chế **liên kết dữ liệu hai chiều (Two-way Data Binding)** được thiết lập một cách bán tự động: Controller được trao quyền kiểm soát bảng (setTableView), trong khi dữ liệu được liên kết phản ứng thông qua setItems. Điều này đảm bảo rằng dù dữ liệu ca làm việc (Shift) thay đổi do tương tác trên giao diện Lịch (Calendar) hay nhập liệu thủ công, TableView này (nếu được hiển thị) sẽ luôn phản ánh trạng thái mới nhất mà không cần logic đồng bộ hóa phức tạp. Cơ chế xử lý sự kiện tại dòng 756-762 cũng tuân theo mẫu **Ủy quyền (Delegation)**, chuyển giao toàn bộ logic điền form cho Controller, đảm bảo tính đóng gói của logic nghiệp vụ.

Tuy nhiên, điểm thú vị nhất nằm ở nhận định tại dòng 764-765: phương thức này tồn tại nhưng **không được gọi** trong luồng chính createShiftTab(). Điều này cho thấy một dấu vết của quá trình phát triển (Development Artifact):

- **Thiết kế song song (Dual View Strategy):** Có thể kiến trúc ban đầu dự định cung cấp hai chế độ xem cho người dùng: chế độ "Lịch" (trực quan) và chế độ "Danh sách" (chi tiết dạng bảng).
- **Mã nguồn dự phòng (Legacy/Fallback Code):** TableView thường dễ debug và kiểm soát hơn CalendarView tùy chỉnh. Việc giữ lại mã nguồn này có thể phục vụ mục đích kiểm thử (testing) hoặc làm phương án dự phòng nếu giao diện Lịch gặp lỗi nghiêm trọng.
- **Tính sẵn sàng mở rộng:** Ứng dụng đã sẵn sàng để thêm tính năng "Switch View" trong tương lai mà không cần viết lại mã nguồn hiển thị danh sách.

3.2.9 Calendar View: Method createCalendarView() Phần 1

Method ‘createCalendarView()’ là một trong những method phức tạp nhất trong ‘MainApp’, tạo giao diện lịch tuần theo phong cách Google Calendar để hiển thị và quản lý ca làm việc. Method này được chia thành nhiều phần để phân tích rõ ràng.

a) Khai báo Method và Container chính

```
    private VBox createCalendarView() {
        VBox calendarBox = new VBox(10);
        calendarBox.setPadding(new Insets(10));
        calendarBox.setStyle("-fx-background-color: white; -fx-
border-color: #ddd; -fx-border-radius: 5;");
        calendarBox.setMinWidth(800);
```

Đoạn mã từ dòng 771 đến 775 thiết lập nền móng cấu trúc cho thành phần giao diện phức tạp nhất của ứng dụng: **Calendar View**. Phương thức này khởi tạo một VBox đóng vai trò là container chính, chịu trách nhiệm chứa thanh điều hướng (header) và lưới hiển thị lịch (grid). Việc tái áp dụng bộ quy tắc CSS "Card Style" (nền trắng, viền xám, bo góc) thể hiện tư duy thiết kế hệ thống (Design System Thinking), đảm bảo sự liền mạch về thị giác khi người dùng chuyển đổi ngữ cảnh từ quản lý danh sách nhân viên sang lập kế hoạch phân ca.

Điểm mấu chốt về kỹ thuật nằm ở dòng 775 với thiết lập setMinWidth(800). Đây là một **ràng buộc bố cục cứng (Hard Layout Constraint)** cần thiết cho loại hình hiển thị này. Khác với TableView có thể co giãn linh hoạt, giao diện lịch phải

hiển thị đồng thời hai chiều dữ liệu: trục ngang (7 ngày trong tuần) và trục dọc (khung thời gian). Nếu không có chiều rộng tối thiểu được đảm bảo, các cột ngày sẽ bị nén lại khiến thông tin trở nên khó đọc, phá vỡ trải nghiệm người dùng. Do đó, dòng lệnh này đóng vai trò "bảo hiểm" cho tính khả dụng (usability) của giao diện trước các thay đổi kích thước cửa sổ.

b) Tạo header

```
    // Header with navigation
    HBox headerBox = new HBox(10);
    headerBox.setAlignment(Pos.CENTER);
    headerBox.setPadding(new Insets(10));

    Button prevWeekButton = new Button("◀ Previous");
    prevWeekButton.setOnAction(e -> {
        currentWeekStart = currentWeekStart.minusWeeks(1);
        refreshCalendarView(calendarBox);
    });

    Button nextWeekButton = new Button("Next ▶");
    nextWeekButton.setOnAction(e -> {
        currentWeekStart = currentWeekStart.plusWeeks(1);
        refreshCalendarView(calendarBox);
    });

    Button todayButton = new Button("Today");
    todayButton.setOnAction(e -> {
        currentWeekStart                                    =
LocalDate.now().with(DayOfWeek.MONDAY);
        refreshCalendarView(calendarBox);
    });

    Label weekLabel = new Label();
    weekLabel.setStyle("-fx-font-size:    18px;    -fx-font-
weight: bold;");
    updateWeekLabel(weekLabel);

    headerBox.getChildren().addAll(prevWeekButton,
todayButton, weekLabel, nextWeekButton);
    HBox.setHgrow(weekLabel, Priority.ALWAYS);
    weekLabel.setAlignment(Pos.CENTER);
```

Đoạn mã từ dòng 778 đến 798 tập trung xây dựng cơ chế **Điều hướng Thời gian (Temporal Navigation)**. Logic xử lý sự kiện tại đây tận dụng tối đa tính năng

của Java Date-Time API. Đặc biệt, cách xử lý nút "Today" (dòng 796-797) thể hiện tư duy lập trình phòng ngừa: thay vì chỉ gọi LocalDate.now(), mã nguồn sử dụng .with(DayOfWeek.MONDAY). Điều này ép buộc ngày hiện tại phải được "quy hoạch" về ngày bắt đầu chuẩn của tuần, đảm bảo lưới lịch luôn hiển thị khung thời gian từ Thứ Hai đến Chủ Nhật, ngăn chặn việc hiển thị các tuần bị lệch (ví dụ: từ Thứ Tư đến Thứ Ba tuần sau). Việc sử dụng các phương thức bất biến (immutable) như minusWeeks và plusWeeks đảm bảo tính toàn vẹn dữ liệu thời gian, tránh các tác dụng phụ không mong muốn (side effects) lên biến trạng thái gốc.

Về mặt bố cục giao diện (Layout Strategy), đoạn mã từ dòng 800 đến 806 áp dụng kỹ thuật **Quản lý Không gian Động**. Bằng cách thiết lập HBox.setHgrow(weekLabel, Priority.ALWAYS) cho nhãn hiển thị ngày, ứng dụng biến thành phần này thành một "lò xo" ảo. Nó sẽ chiếm dụng toàn bộ không gian trống còn thừa trong thanh tiêu đề, đẩy các nút điều hướng (Previous, Today, Next) dạt về hai phía hoặc giữ khoảng cách cân đối tùy theo kích thước cửa sổ. Kỹ thuật này giúp giao diện duy trì tính thẩm mỹ và cân đối (responsive) ngay cả khi người dùng thay đổi kích thước ứng dụng, đồng thời giữ cho thông tin quan trọng nhất (khoảng thời gian hiển thị) luôn nằm ở trọng tâm thị giác.

c) Tạo Calendar Grid và Row Constraints

```
    // Calendar grid
    GridPane calendarGrid = new GridPane();
    calendarGrid.setHgap(1);
    calendarGrid.setVgap(1);
    calendarGrid.setStyle("-fx-background-color: #e0e0e0;");

    // Initialize row constraints for all time slots (6 AM to
11 PM = 18 rows)
    for (int i = 0; i < 19; i++) {
        RowConstraints rowConstraint = new RowConstraints();
        rowConstraint.setMinHeight(50);
        rowConstraint.setPrefHeight(50);
        rowConstraint.setMaxHeight(50);
        calendarGrid.getRowConstraints().add(rowConstraint);
    }
```

Đoạn mã từ dòng 809 đến 820 thiết lập nền tảng cấu trúc không gian cho Calendar View thông qua đối tượng GridPane. Việc lựa chọn GridPane là quyết định

kiến trúc tối ưu cho bài toán này, bởi khả năng hỗ trợ **Row/Column Spanning** (trải dài qua nhiều hàng/cột)—tính năng cốt lõi để hiển thị các ca làm việc kéo dài nhiều giờ liền mạch. Về mặt thị giác, dòng 810-812 áp dụng một thủ pháp đồ họa kinh điển (CSS Trick): thay vì vẽ viền cho từng ô (gây nặng nề cho việc render), mã nguồn thiết lập khoảng hở (gap) 1 pixel kết hợp với màu nền xám. Màu nền này sẽ "lộ" ra qua các khe hở, tạo hiệu ứng đường kẻ lưới sắc nét và hiệu quả về mặt hiệu năng.

Trọng tâm logic nằm ở vòng lặp thiết lập RowConstraints (dòng 815-820). Tại đây, ứng dụng xây dựng một **Hệ tọa độ tuyến tính (Linear Coordinate System)** cố định: mỗi đơn vị thời gian (1 giờ) được quy đổi chính xác thành đơn vị không gian (50 pixel).

- **Tính bất biến:** Việc khóa chặt chiều cao (Min, Pref, Max đều bằng 50) ngăn chặn cơ chế tự động co giãn của JavaFX.
- **Ý nghĩa:** Điều này biến lưới hiển thị thành một trục thời gian chuẩn xác. Nhờ đó, logic hiển thị sau này trở nên đơn giản: một ca làm 4 tiếng sẽ luôn chiếm đúng 4 hàng (200px), đảm bảo tính trực quan và chính xác về tỷ lệ thời gian cho người quản lý.

d) Tạo Day Header

```
    // Day headers ensure equal width for all 7 days
    String[] dayNames = {"Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"};
    for (int i = 0; i < 7; i++) {
        VBox dayHeader = new VBox(5);
        dayHeader.setAlignment(Pos.CENTER);
        dayHeader.setPadding(new Insets(10));
        dayHeader.setStyle("-fx-background-color: #4285f4;");
        dayHeader.setMaxWidth(Double.MAX_VALUE);

        Label dayNameLabel = new Label(dayNames[i]);
        dayNameLabel.setStyle("-fx-font-weight: bold; -fx-text-
fill: white; -fx-font-size: 12px;");

        LocalDate dayDate = currentWeekStart.plusDays(i);
        Label dateLabel = new
Label(dayDate.format(DateTimeFormatter.ofPattern("MMM d")));
        dateLabel.setStyle("-fx-text-fill: white; -fx-font-size:
14px;");

        dayHeader.getChildren().addAll(dayNameLabel, dateLabel);
```

```
            GridPane.setHgrow(dayHeader, Priority.ALWAYS);
            calendarGrid.add(dayHeader, i + 1, 0);
        }
```

Đoạn mã từ dòng 824 đến 839 tập trung vào việc xây dựng hàng tiêu đề trực quan, đóng vai trò định hướng trục ngang cho toàn bộ giao diện lịch. Việc khởi tạo một vòng lặp cố định 7 lần (tương ứng 7 ngày trong tuần) kết hợp với mảng dayNames tĩnh giúp xác định cấu trúc tuần tự rõ ràng. Về mặt thiết kế trải nghiệm người dùng (UX), việc áp dụng màu nền xanh #4285f4 (dòng 829) không phải là ngẫu nhiên; đây là sự vay mượn có chủ đích từ ngôn ngữ thiết kế của Google Calendar, tạo cảm giác quen thuộc và chuyên nghiệp. Đồng thời, logic tính toán ngày tháng tại dòng 835 (currentWeekStart.plusDays(i)) đảm bảo tính chính xác tuyệt đối của dữ liệu thời gian: mỗi cột được gắn chặt với một ngày cụ thể trong mô hình dữ liệu, chứ không chỉ là nhãn văn bản tĩnh.

Về kỹ thuật dàn trang (Layout Engineering), dòng 840 và 841 thiết lập quy tắc không gian cốt lõi cho GridPane. Lệnh GridPane.setHgrow(..., Priority.ALWAYS) ép buộc cả 7 cột ngày phải chia sẻ đều không gian chiều ngang, bất kể kích thước cửa sổ, đảm bảo sự cân đối. Quan trọng hơn, việc thêm các header vào tọa độ cột i + 1 (dòng 841) thể hiện tư duy quy hoạch lưới rõ ràng: Cột có chỉ số 0 được chủ động "để dành" (reserved) cho trục thời gian (Time Axis). Điều này thiết lập hệ tọa độ tiêu chuẩn cho toàn bộ lưới: Trục X (cột 1-7) đại diện cho Ngày, và Trục Y (hàng 0-18) đại diện cho Giờ, tạo nền tảng vững chắc để định vị chính xác các ô ca làm việc (Shift cells) sau này.

e) Tính toán Covered Hours

```
        // Pre-calculate which hours are covered by spanning shifts for each day
        Map<LocalDate, Set<Integer>> coveredHours = new HashMap<>();
        for (int day = 0; day < 7; day++) {
            LocalDate dayDate = currentWeekStart.plusDays(day);
            List<Shift>                    allShiftsForDate                    =
shiftController.getShiftsByDate(dayDate);
            Set<Integer> covered = new HashSet<>();

            for (Shift shift : allShiftsForDate) {
                if (shift.getStartTime() != null && shift.getEndTime() != null) {
                    int startHour = shift.getStartTime().getHour();
                    int startMinute = shift.getStartTime().getMinute();
                    int endHour = shift.getEndTime().getHour();
                    int endMinute = shift.getEndTime().getMinute();

                    int startTotalMinutes = startHour * 60 + startMinute;
```

```
                    int endTotalMinutes = endHour * 60 + endMinute;
                    int durationMinutes = endTotalMinutes startTotalMinutes;
                    int  rowSpan  =  Math.max(1,  (int)  Math.ceil(durationMinutes  /
60.0));

                    // Mark all hours covered by this shift (except the starting
hour)
                    for (int h = 1; h < rowSpan; h++) {
                        int coveredHour = startHour + h;
                        if (coveredHour < 24) {
                            covered.add(coveredHour);
                        }
                    }
                }
            }
            coveredHours.put(dayDate, covered);
        }
```

Đoạn mã từ dòng 844 đến 872 thực hiện một bước **tiền xử lý dữ liệu (Data Pre-processing)** cực kỳ quan trọng cho GridPane. Vì GridPane không tự động ngăn chặn các ô (cell) đè lên nhau, ứng dụng cần tự xác định trước những vị trí sẽ bị chiếm dụng bởi các ca làm việc kéo dài (multi-hour shifts). Cấu trúc Map<LocalDate, Set<Integer>> hoạt động như một "bản đồ nhiệt" (heatmap): nó đánh dấu các tọa độ thời gian (ngày, giờ) đã có người "đặt chỗ". Việc sử dụng shiftController.getShiftsByDate(dayDate) tại dòng 848 một lần nữa khẳng định sự phụ thuộc của View vào Controller để lấy dữ liệu thô, sau đó View tự xử lý logic hiển thị mà không làm ô nhiễm tầng dữ liệu.

Trọng tâm thuật toán nằm ở việc chuyển đổi dữ liệu thời gian liên tục thành các đơn vị lưới rời rạc (dòng 858-869). Phép toán Math.ceil(durationMinutes / 60.0) đảm bảo rằng một ca làm việc dù chỉ lấn sang giờ tiếp theo 1 phút cũng sẽ chiếm trọn hàng đó, giúp hiển thị trực quan không bị cắt xén. Vòng lặp từ dòng 864 có logic rất đặc thù: nó bắt đầu từ h = 1 (bỏ qua giờ khởi đầu). Điều này là do giờ khởi đầu là nơi Component ca làm việc thực sự được vẽ; các giờ tiếp theo (coveredHour) là các "ô bóng ma" nằm bên dưới rowSpan. Chúng cần được đánh dấu vào Set để trong quá trình render lưới sau này (ở các dòng tiếp theo), ứng dụng biết đường **bỏ qua**, không vẽ các ô trống mặc định vào vị trí đó nữa, tránh tình trạng xung đột hiển thị (UI Collision) hoặc render đè lên nhau.

3.2.10 Calendar View: Method createCalendarView() Phần 2 (Time Slots)

Phần tiếp theo của 'createCalendarView()' tạo các ô thời gian trong grid và xử lý hiển thị ca làm việc.

a) Tạo Time Label Column và Day Columns

```
// Time slots (6 AM to 11 PM) ensure full week is visible
for (int hour = 6; hour < 24; hour++) {
    // Time label column
    Label timeLabel = new Label(String.format("%02d:00", hour));
    timeLabel.setStyle("-fx-font-size: 11px; -fx-padding: 5px;");
    timeLabel.setPrefWidth(60);
    timeLabel.setMinWidth(60);
    timeLabel.setAlignment(Pos.CENTER_RIGHT);
    calendarGrid.add(timeLabel, 0, hour 5);

    // Day columns ensure equal width
    for (int day = 0; day < 7; day++) {
        final LocalDate slotDate = currentWeekStart.plusDays(day);
        final int hourSlot = hour;

        // Check if this hour is already covered by a shift from a previous hour
        Set<Integer> covered = coveredHours.getOrDefault(slotDate, Collections.emptySet());
        boolean isCovered = covered.contains(hourSlot);

        // Get all shifts for this date
        List<Shift> allShiftsForDate = shiftController.getShiftsByDate(slotDate);

        // Find shifts that START in this hour slot (always show shifts in their starting hour)
        List<Shift> shiftsStartingInSlot = allShiftsForDate.stream()
                .filter(s -> s.getStartTime() != null && s.getEndTime() != null &&
                        s.getStartTime().getHour() == hourSlot)
                .collect(Collectors.toList());
```

Đoạn mã từ dòng 875 đến 883 thiết lập **Trục Thời Gian (Time Axis)**, đóng vai trò là xương sống định vị cho toàn bộ giao diện lịch. Vòng lặp ngoài khởi tạo một hệ tọa độ tuyến tính từ 6:00 đến 23:00, trong đó mỗi giờ tương ứng với một hàng trên GridPane. Kỹ thuật định dạng chuỗi %02d:00 và căn lề phải (Pos.CENTER_RIGHT) không chỉ đảm bảo tính thẩm mỹ mà còn giúp người dùng dễ dàng đối chiếu thời gian. Đáng chú ý là công thức ánh xạ tọa độ calendarGrid.add(timeLabel, 0, hour 5): phép toán trừ đi 5 đơn vị là sự chuyển đổi cần thiết giữa không gian dữ liệu (bắt đầu từ giờ thứ 6) và không gian hiển thị (bắt đầu từ hàng thứ 1, nhường hàng 0 cho tiêu đề), đảm bảo dữ liệu được đặt đúng vị trí logic trên lưới.

Trong vòng lặp lồng nhau xử lý từng ngày (dòng 886-901), logic chương trình chuyển sang giải quyết bài toán **Phân loại và Lọc dữ liệu**. Việc khai báo biến final cho slotDate và hourSlot là bắt buộc để thỏa mãn quy tắc bao đóng (closure) của Java khi sử dụng trong biểu thức Lambda. Cơ chế kiểm tra coveredHours (dòng 891-892) hoạt động như một chốt chặn logic, xác định xem ô lưới hiện tại là một "khoảng trống" hay là "phần mở rộng" của một ca làm việc từ giờ trước. Đặc biệt, việc sử dụng **Stream API** tại dòng 898-901 thay thế cho vòng lặp truyền thống giúp mã nguồn trở

nên gãy gọn và mang tính khai báo (declarative): hệ thống chỉ trích xuất các ca làm việc có thời gian *bắt đầu* trùng khớp chính xác với hourSlot. Điều này đảm bảo mỗi ca làm việc chỉ được khởi tạo UI một lần duy nhất tại điểm bắt đầu, tránh trùng lặp đối tượng hiển thị.

b) Tạo Time Slot Container

```
        // Create time slot container
        // Use StackPane to allow overlapping shifts if needed
        StackPane timeSlot = new StackPane();

        // If this hour is covered by a spanning shift, make it
transparent
        if (isCovered) {
            timeSlot.setStyle("-fx-background-color: transparent; -
fx-border-color: transparent;");
        } else {
            timeSlot.setStyle("-fx-background-color: white; -fx-
border-color: #e0e0e0;");
        }
        timeSlot.setPadding(new Insets(2));
        // Default height for single hour
        timeSlot.setPrefHeight(50);
        timeSlot.setMinHeight(50);
```

Đoạn mã từ dòng 903 đến 916 tập trung vào việc khởi tạo đơn vị hiển thị cơ bản của lưới lịch thông qua StackPane. Việc lựa chọn container này (dòng 905) thay vì Pane thông thường phản ánh tư duy thiết kế linh hoạt: cơ chế xếp chồng (stacking) của nó cho phép dễ dàng mở rộng trong tương lai, chẳng hạn như hiển thị các huy hiệu trạng thái (badges) hoặc cảnh báo xung đột đè lên trên ô lịch mà không làm vỡ bố cục. Thiết lập chiều cao cố định 50px (dòng 915-916) đảm bảo sự đồng bộ tuyệt đối với RowConstraints đã định nghĩa trước đó, duy trì tính toàn vẹn hình học của lưới.

Điểm tinh tế về mặt kỹ thuật nằm ở logic xử lý điều kiện hiển thị tại dòng 908-911. Đây là mảnh ghép cuối cùng để hoàn thiện tính năng hiển thị ca làm việc kéo dài (multi-hour shift). Đối với các ô được đánh dấu là isCovered (tức là nằm bên dưới một ca làm việc đã bắt đầu từ giờ trước), ứng dụng áp dụng kiểu hiển thị "trong suốt" (transparent). Kỹ thuật này biến ô lưới thành một "bóng ma" vô hình: nó vẫn chiếm giữ không gian vật lý trong GridPane để duy trì cấu trúc, nhưng nhường hoàn toàn không gian thị giác cho khối ca làm việc (rowSpan) đang tràn xuống từ trên. Ngược

lại, các ô không bị che phủ sẽ được render đầy đủ viền và nền, tạo nên các khoảng trống trực quan (Visual Affordance) mời gọi người dùng thực hiện thao tác thêm mới.

c) Hiển thị Shift Blocks trong Time Slot

```
      // Only add shift content if shifts start here AND this hour is not
covered
      if (!shiftsStartingInSlot.isEmpty() && !isCovered) {
          int maxRowSpan = 1;

          // Calculate max row span first
          for (Shift shift : shiftsStartingInSlot) {
              int startHour = shift.getStartTime().getHour();
              int startMinute = shift.getStartTime().getMinute();
              int endHour = shift.getEndTime().getHour();
              int endMinute = shift.getEndTime().getMinute();

              int startTotalMinutes = startHour * 60 + startMinute;
              int endTotalMinutes = endHour * 60 + endMinute;
              int durationMinutes = endTotalMinutes startTotalMinutes;

              // Calculate how many hour slots to span (rounded up)
              int rowSpan = Math.max(1, (int) Math.ceil(durationMinutes /
60.0));
              maxRowSpan = Math.max(maxRowSpan, rowSpan);
          }
```

Đoạn mã từ dòng 918 đến 935 đóng vai trò là bộ não tính toán bố cục (Layout Calculation Engine) cho mỗi ô lưới. Điều kiện kiểm tra tại dòng 919 (!shiftsStartingInSlot.isEmpty() && !isCovered) hoạt động như một bộ lọc kết xuất (rendering filter) quan trọng: nó đảm bảo rằng khối ca làm việc (Shift Block) chỉ được khởi tạo duy nhất tại **điểm bắt đầu** của dòng thời gian. Nếu không có bộ lọc !isCovered, hệ thống sẽ cố gắng vẽ lại nội dung đè lên các ô đang bị chiếm dụng bởi một ca làm việc dài từ giờ trước, gây ra lỗi hiển thị chồng chéo.

Trọng tâm xử lý nằm ở thuật toán tìm maxRowSpan (dòng 920-935). Trong trường hợp (dù hiếm gặp) có nhiều ca làm việc bắt đầu cùng một giờ, ứng dụng cần xác định ca nào có thời lượng dài nhất để thiết lập chiều cao cho container chứa. Việc chuyển đổi thời gian sang không gian lại được thực hiện: từ hiệu số phút (durationMinutes) sang số lượng hàng lưới (rowSpan). Bằng cách tính toán trước maxRowSpan trước khi thực sự thêm bất kỳ thành phần giao diện con nào, ứng dụng đảm bảo rằng StackPane cha sẽ có đủ không gian vật lý để "ôm" trọn vẹn ca làm việc

dài nhất, tránh tình trạng nội dung bị cắt cụt (clipping) hoặc tràn ra ngoài ô lưới một cách mất kiểm soát.

d) Thiết lập chiều cao Container và tạo Shifts Container

```
// Set container height BEFORE adding children
timeSlot.setPrefHeight(50 * maxRowSpan);
timeSlot.setMinHeight(50 * maxRowSpan);
timeSlot.setMaxHeight(50 * maxRowSpan);

// Use VBox inside StackPane for vertical stacking
VBox shiftsContainer = new VBox(2);
shiftsContainer.setMaxWidth(Double.MAX_VALUE);
shiftsContainer.setMaxHeight(Double.MAX_VALUE);
```

Bằng cách thiết lập cứng chiều cao container theo công thức 50 * maxRowSpan, ứng dụng chuyển đổi trực tiếp đại lượng thời gian (giờ) thành đại lượng không gian (pixel). Việc khóa chặt cả ba thuộc tính Min, Pref, và Max Height là một kỹ thuật quan trọng trong JavaFX để "vô hiệu hóa" cơ chế tự động co giãn của Layout Engine, ép buộc giao diện phải tuân thủ tuyệt đối cấu trúc lưới đã định nghĩa, đảm bảo các ca làm việc dài (ví dụ: 4 tiếng) chiếm đúng không gian vật lý của 4 ô lưới dọc.

Tiếp theo, việc lồng ghép một VBox vào bên trong StackPane (dòng 943-946) thể hiện giải pháp xử lý trường hợp **Đồng quy (Concurrency Handling)**. Trong tình huống có nhiều nhân viên cùng bắt đầu ca làm việc tại một giờ cụ thể (ví dụ: 3 người cùng làm ca sáng lúc 8:00), StackPane đơn thuần sẽ xếp chồng họ lên nhau theo trục Z, làm che khuất nội dung. VBox giải quyết vấn đề này bằng cách sắp xếp các ca làm việc theo trục dọc (Y-axis) bên trong ô thời gian đó. Thiết lập setMaxWidth/Height với Double.MAX_VALUE biến VBox này thành một container linh hoạt, tự động mở rộng để chiếm trọn vẹn không gian đã được "quy hoạch" bởi StackPane cha, sẵn sàng tiếp nhận và hiển thị danh sách các ca làm việc một cách ngăn nắp.

e) Tạo Shift Label cho từng ca làm việc

```
for (Shift shift : shiftsStartingInSlot) {
    // Calculate duration in hours (rounded up)
    int startHour = shift.getStartTime().getHour();
    int startMinute = shift.getStartTime().getMinute();
    int endHour = shift.getEndTime().getHour();
    int endMinute = shift.getEndTime().getMinute();

    // Calculate total minutes
```

```
            int startTotalMinutes = startHour * 60 + startMinute;
            int endTotalMinutes = endHour * 60 + endMinute;
            int durationMinutes = endTotalMinutes startTotalMinutes;

            // Calculate how many hour slots to span (minimum 1, rounded up)
            int rowSpan = Math.max(1, (int) Math.ceil(durationMinutes /
60.0));

            // Create shift label
            Label shiftLabel = new Label(shift.getEmployeeName() + "\n" +
                String.format("%02d:%02d-%02d:%02d",
                    shift.getStartTime().getHour(),
shift.getStartTime().getMinute(),
                    shift.getEndTime().getHour(),
shift.getEndTime().getMinute()));
            shiftLabel.setStyle("-fx-font-size: 9px; -fx-padding: 2px; -fx-
background-color: #4285f4; -fx-text-fill: white; -fx-background-radius:
3px;");
            shiftLabel.setMaxWidth(Double.MAX_VALUE);
            shiftLabel.setWrapText(true);
            shiftLabel.setAlignment(Pos.TOP_LEFT);

            // Set height based on duration (each hour slot is 50px)
            // If multiple shifts start at same time, stack them vertically
            double heightPerShift = (50 * rowSpan 4) /
shiftsStartingInSlot.size();
            shiftLabel.setPrefHeight(heightPerShift);
            shiftLabel.setMinHeight(heightPerShift);
            shiftLabel.setMaxHeight(heightPerShift);
```


Đoạn mã từ dòng 948 đến 971 tập trung vào quá trình **biểu diễn thông tin (Information Visualization)**, chuyển đổi dữ liệu thô thành các thẻ UI trực quan. Logic tạo nội dung tại dòng 964-967 ưu tiên hiển thị Tên nhân viên và Khung giờ—những thông tin cốt lõi nhất cho người quản lý lịch trình. Về mặt thẩm mỹ, việc áp dụng hệ thống định dạng "Card UI" với nền xanh dương (#4285f4), chữ trắng và bo góc (dòng 968) không chỉ tạo độ tương phản cao giúp dễ đọc mà còn gợi nhớ đến giao diện quen thuộc của Google Calendar, giảm thiểu đường cong học tập cho người dùng. Các thiết lập layout như setMaxWidth(Double.MAX_VALUE) và setWrapText(true) đảm bảo thẻ tự động thích ứng với không gian ngang, giữ cho nội dung không bị cắt xén dù tên nhân viên dài.

Điểm nhấn kỹ thuật nằm ở thuật toán **Phân chia không gian động (Dynamic Space Partitioning)** tại dòng 973-978. Đây là chiến lược giải quyết xung đột hiển thị (Conflict Resolution) khi có nhiều ca làm việc bắt đầu cùng một giờ (Concurrency). Thay vì chồng đè lên nhau gây rối mắt, ứng dụng tính toán tổng chiều cao khả dụng

dựa trên thời lượng ca (50 * rowSpan) và chia đều cho số lượng ca đồng thời (/ size). Điều này có nghĩa là nếu có 2 nhân viên cùng bắt đầu làm việc trong một ca dài 4 tiếng, chiều cao thẻ của mỗi người sẽ được chia sẻ để cùng nằm gọn trong khối thời gian đó. Cách tiếp cận này hy sinh một phần chiều cao của từng thẻ riêng lẻ để đổi lấy một bố cục tổng thể ngăn nắp, đảm bảo mọi ca làm việc đều được hiển thị mà không phá vỡ cấu trúc lưới thời gian.

f) Event Handler cho Shift Label

```
shiftLabel.setOnMouseClicked(e -> {
    datePicker.setValue(slotDate);
    // Find and select the shift in the table
    if (shiftTable != null) {
        shiftTable.getSelectionModel().select(shift);
        shiftController.handleTableSelection();
    }
});
```

Đoạn mã từ dòng 980 đến 985 đóng vai trò thiết lập **cơ chế phản hồi tương tác (Interactive Feedback Loop)**, biến giao diện Lịch từ một bảng hiển thị tĩnh thành một công cụ quản lý năng động. Trọng tâm của logic này là sự **đồng bộ hóa ngữ cảnh (Context Synchronization)**: ngay khi sự kiện MouseClicked được kích hoạt, hệ thống lập tức cập nhật trạng thái của DatePicker và cố gắng đồng bộ lựa chọn sang mô hình của TableView (thông qua selectionModel). Điều này đảm bảo tính nhất quán của trạng thái ứng dụng: dù người dùng đang nhìn vào giao diện đồ họa (Lịch) hay danh sách (Bảng), dữ liệu được chọn luôn là một, tạo tiền đề cho các thao tác chỉnh sửa chính xác.

Đáng chú ý nhất là việc tái sử dụng phương thức shiftController.handleTableSelection() tại dòng 985. Thay vì viết lại logic điền form (form population logic) riêng biệt cho Calendar View, mã nguồn tuân thủ triệt để nguyên tắc **DRY (Don't Repeat Yourself)** và mẫu thiết kế **Ủy quyền (Delegation)**. View chỉ đóng vai trò phát hiện sự kiện, còn việc trích xuất dữ liệu từ đối tượng Shift và đẩy vào các trường nhập liệu được giao phó hoàn toàn cho Controller. Cách tiếp cận này tạo ra trải nghiệm "Click-to-Edit" (Nhấp để sửa) liền mạch: người dùng chỉ cần chọn một ca trên lịch, và form bên cạnh sẽ tự động điền đầy đủ thông tin để sẵn sàng cập nhật.

g) Hoàn thiện Time Slot và thêm vào Grid

```
// Add to shifts container
shiftsContainer.getChildren().add(shiftLabel);
}

// Add shifts container to time slot
timeSlot.getChildren().add(shiftsContainer);
StackPane.setAlignment(shiftsContainer, Pos.TOP_LEFT);

// Set row span for the time slot container to span multiple rows
// This must be set BEFORE adding to grid
if (maxRowSpan > 1) {
    GridPane.setRowSpan(timeSlot, maxRowSpan);
}
```

Đoạn mã từ dòng 990 đến 994 hoàn tất quy trình lắp ráp cấu trúc phân cấp (Hierarchical Assembly) của ô lịch. Bằng việc lồng shiftsContainer (VBox) vào timeSlot (StackPane) với thiết lập căn chỉnh Pos.TOP_LEFT, ứng dụng đảm bảo tính chính xác tuyệt đối về mặt hiển thị thời gian: nội dung luôn bắt đầu từ mốc giờ khởi đầu của ô lưới. Cấu trúc lồng nhau chặt chẽ này (StackPane chứa VBox, VBox chứa Labels) thể hiện sự phân chia trách nhiệm rõ ràng: StackPane chịu trách nhiệm định vị ô trong lưới tọa độ tổng thể, trong khi VBox đóng vai trò quản lý trật tự nội dung bên trong, giải quyết gọn gàng bài toán hiển thị các ca làm việc đồng thời (Concurrency) mà không làm vỡ bố cục chung.

Điểm nhấn kỹ thuật quan trọng nhất nằm ở dòng 999-1001 với lệnh GridPane.setRowSpan. Đây là bước "chốt hạ" chuyển giao kết quả tính toán logic sang chỉ thị hiển thị cho Layout Engine của JavaFX. Việc kiểm tra điều kiện maxRowSpan > 1 trước khi áp dụng thuộc tính là một kỹ thuật tối ưu hóa hiệu năng (Performance Optimization), loại bỏ các overhead không cần thiết cho đa số các ca làm việc tiêu chuẩn (1 giờ). Nhờ cơ chế này, các ca làm việc kéo dài sẽ được render thành một khối hình học thống nhất, liền mạch trải dài qua nhiều hàng, thay vì bị đứt gãy thành các ô rời rạc, giúp người dùng hình dung trực quan về độ dài của ca làm việc.

h) Hoàn thiện Calendar View

```
// If no shifts start here, timeSlot remains empty (default state)

GridPane.setHgrow(timeSlot, Priority.ALWAYS);

// Add to grid
int gridRow = hour 5;
```

```
        calendarGrid.add(timeSlot, day + 1, gridRow);
        }
        }

        ScrollPane scrollPane = new ScrollPane(calendarGrid);
        scrollPane.setFitToWidth(true);
        scrollPane.setFitToHeight(true);
        scrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS_NEEDED);
        scrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.NEVER);

        calendarBox.getChildren().addAll(headerBox, scrollPane);
        VBox.setVgrow(scrollPane, Priority.ALWAYS);

        return calendarBox;
        }
```

Đoạn mã từ dòng 1003 đến 1022 đánh dấu giai đoạn **Lắp ráp và Hoàn thiện (Assembly & Finalization)** của module hiển thị lịch. Sau khi logic phức tạp về tính toán vị trí và xử lý chồng chéo (concurrency) đã hoàn tất, mã nguồn tập trung vào việc định hình hành vi tương tác của container. Dòng 1005 với GridPane.setHgrow(..., Priority.ALWAYS) là chốt chặn cuối cùng cho tính đáp ứng (responsiveness) của lưới: nó ép buộc mọi ô thời gian—dù chứa dữ liệu hay trống rỗng—phải tuân thủ kỷ luật phân chia không gian ngang, đảm bảo 7 cột ngày luôn có chiều rộng bằng nhau và co giãn linh hoạt theo cửa sổ ứng dụng.

Điểm nhấn kỹ thuật quan trọng nhất nằm ở việc tích hợp ScrollPane (dòng 1013-1017). Đây là giải pháp bắt buộc cho vấn đề **Tràn nội dung (Content Overflow)**: với 18 hàng giờ (mỗi hàng 50px) cộng thêm header, chiều cao thực tế của lưới vượt quá 900px, lớn hơn chiều cao hiển thị mặc định (thường là 600-800px).

- **Cuộn dọc (AS_NEEDED):** Chỉ xuất hiện khi cần thiết, giữ giao diện sạch sẽ.
- **Cuộn ngang (NEVER):** Kết hợp với setFitToWidth(true), đây là một quyết định UX tinh tế. Nó ép buộc nội dung phải co lại cho vừa chiều ngang, ngăn chặn thanh cuộn ngang xuất hiện—một yếu tố thường gây khó chịu khi xem lịch trình.

Cuối cùng, dòng 1019-1022 hoàn tất cấu trúc phân cấp (Hierarchy): Header (cố định) và ScrollPane (chứa Grid cuộn được) được đóng gói vào VBox chính. Lệnh VBox.setVgrow đảm bảo vùng lịch luôn chiếm dụng tối đa không gian màn hình còn lại, đẩy giao diện đến trạng thái "Tràn màn hình" (Full-height layout) chuyên nghiệp.

3.2.11 Helper Methods cho Calendar View

a) Method refreshCalendarView()

```
/**
 * Refresh the calendar view.
 */
private void refreshCalendarView(VBox calendarBox) {
    if (calendarBox != null) {
        calendarBox.getChildren().clear();
        VBox newCalendar = createCalendarView();
        calendarBox.getChildren().addAll(newCalendar.getChildren());
    }
}
```

Đoạn mã từ dòng 1028 đến 1032 thực hiện pattern **"Xóa và Vẽ lại toàn bộ" (Flush and Redraw)**. Đây là một quyết định kỹ thuật đánh đổi giữa **Hiệu năng (Performance)** và **Độ tin cậy (Reliability)**.

- **Đơn giản hóa Quản lý Trạng thái:** Trong các giao diện phức tạp như Lịch (nơi các ô có thể gộp dòng, thay đổi kích thước, hoặc chồng lên nhau), việc cập nhật "tăng dần" (incremental update) — tức là chỉ tìm và sửa các ô thay đổi — là cực kỳ rủi ro và dễ sinh lỗi (bug-prone). Lập trình viên sẽ phải theo dõi ID của từng node, xử lý logic xóa các node cũ bị thừa, thêm node mới... Phương pháp clear() và createCalendarView() đảm bảo rằng giao diện **luôn luôn đồng bộ tuyệt đối** với dữ liệu hiện tại (currentWeekStart). Không bao giờ có chuyện "rác" giao diện (ghost elements) còn sót lại từ tuần trước.
- **Chi phí Hiệu năng (Performance Cost):** Mặc dù việc hủy và tạo mới hàng trăm đối tượng Node (Label, StackPane, VBox) nghe có vẻ nặng nề, nhưng với quy mô của một lưới lịch tuần (7 cột x 18 hàng = 126 ô cơ sở), JavaFX Scenegraph có thể xử lý việc này trong tích tắc (vài mili-giây). Do đó, độ trễ là không đáng kể đối với mắt người dùng.
- **Cơ chế Garbage Collection:** Khi calendarBox.getChildren().clear() được gọi, tất cả các node cũ (của tuần cũ) sẽ bị ngắt tham chiếu khỏi Scene Graph. Bộ thu gom rác (Garbage Collector) của Java sẽ tự động giải phóng bộ nhớ của chúng, đảm bảo ứng dụng không bị rò rỉ bộ nhớ (memory leak) dù người dùng bấm chuyển tuần liên tục.

b) Method updateWeekLabel()

```
    /**
     * Update week label.
     */
    private void updateWeekLabel(Label weekLabel) {
        LocalDate weekEnd = currentWeekStart.plusDays(6);
        String                    weekText                    =
currentWeekStart.format(DateTimeFormatter.ofPattern("MMM d"))
+ " " +
                            weekEnd.format(DateTimeFormatter.o
fPattern("MMM d, yyyy"));
        weekLabel.setText(weekText);
    }
```

Đoạn mã từ dòng 1039 đến 1043, dù ngắn gọn, nhưng đóng vai trò quan trọng trong việc **định hướng ngữ cảnh thời gian (Temporal Context)** cho người dùng.

- **Logic tính toán (Dòng 1040):** Việc sử dụng plusDays(6) là chính xác về mặt toán học lịch để xác định phạm vi tuần (Tuần bắt đầu từ Thứ Hai + 6 ngày = Kết thúc vào Chủ Nhật). Đây là logic bất biến trong các hệ thống lịch chuẩn ISO-8601.
- **Định dạng chuỗi (String Interpolation Dòng 1041-1042):** Ứng dụng chọn cách hiển thị tinh gọn: chỉ hiển thị Năm (Year) ở cuối chuỗi. Điều này giúp giảm thiểu sự lặp lại không cần thiết (ví dụ: thay vì "Jan 1, 2024 Jan 7, 2024" thì chỉ cần "Jan 1 Jan 7, 2024"), giúp giao diện thoáng hơn và người dùng dễ nắm bắt thông tin trọng tâm.
- **Nguyên tắc thiết kế:** Việc tách logic này ra method riêng (updateWeekLabel) thay vì nhúng trực tiếp vào các event handler của nút Next/Prev là minh chứng rõ ràng cho nguyên tắc **Single Responsibility Principle (SRP)**. Nó giúp mã nguồn dễ bảo trì và đảm bảo tính nhất quán định dạng ở mọi nơi.

3.2.12 Phần Form Shift: Method createShiftFormSection()

Method này tạo form nhập liệu cho ca làm việc, bao gồm các trường để chọn nhân viên, ngày, thời gian bắt đầu/kết thúc, và loại ca.

a) Tạo Container và Form Fields Cơ bản

```
    /**
```

```
      * Create the shift form section with hour/minute combo boxes.
      */
     private VBox createShiftFormSection() {
         VBox formBox = new VBox(15);
         formBox.setPadding(new Insets(10));
         formBox.setStyle("-fx-background-color:    white;    -fx-
border-color: #ddd; -fx-border-radius: 5;");

         Label formLabel = new Label("Shift Assignment Form");
         formLabel.setStyle("-fx-font-size: 16px; -fx-font-weight:
bold;");

         // Form fields
         shiftIdField = new TextField();
         shiftIdField.setPromptText("ID (auto-generated)");
         shiftIdField.setEditable(false);
         shiftIdField.setStyle("-fx-background-color: #f0f0f0;");

         employeeComboBox = new ComboBox<>();
         employeeComboBox.setPromptText("Select Employee *");
         employeeComboBox.setPrefWidth(Double.MAX_VALUE);

         datePicker = new DatePicker();
         datePicker.setPromptText("Select Date *");
         datePicker.setValue(LocalDate.now());

         // Start time Hour and Minute
         HBox startTimeBox = new HBox(5);
         startTimeBox.setAlignment(Pos.CENTER_LEFT);
         Label startTimeLabel = new Label("Start Time:");
         startHourComboBox = new ComboBox<>();
         startHourComboBox.setPrefWidth(80);
         Label colon1 = new Label(":");
         startMinuteComboBox = new ComboBox<>();
         startMinuteComboBox.setPrefWidth(80);
         startTimeBox.getChildren().addAll(startTimeLabel,
startHourComboBox, colon1, startMinuteComboBox);

         // End time Hour and Minute
         HBox endTimeBox = new HBox(5);
         endTimeBox.setAlignment(Pos.CENTER_LEFT);
         Label endTimeLabel = new Label("End Time:");
         endHourComboBox = new ComboBox<>();
         endHourComboBox.setPrefWidth(80);
         Label colon2 = new Label(":");
         endMinuteComboBox = new ComboBox<>();
         endMinuteComboBox.setPrefWidth(80);
         endTimeBox.getChildren().addAll(endTimeLabel,
endHourComboBox, colon2, endMinuteComboBox);
```

```
        shiftTypeComboBox = new ComboBox<>();
        shiftTypeComboBox.setPromptText("Select Shift Type *");
        shiftTypeComboBox.setPrefWidth(Double.MAX_VALUE);
```

Đoạn mã từ dòng 1049 đến 1095 tập trung vào việc xây dựng **Giao diện Nhập liệu (Input Interface)**. Điểm thú vị nhất ở đây là cách lập trình viên giải quyết bài toán "Chọn giờ" trong JavaFX tiêu chuẩn (Standard JavaFX).

Vì thư viện JavaFX gốc không có component TimePicker chuyên dụng (như DatePicker), lập trình viên đã áp dụng giải pháp **"Chia để trị" (Divide and Conquer)**: tách thời gian thành hai trường ComboBox riêng biệt cho Giờ và Phút.

- **Chiến lược Ràng buộc Cứng (Hard Constraints):** Thay vì dùng TextField và buộc người dùng nhập "08:30" (dễ gây lỗi format, ví dụ nhập "8h30", "8.30" hoặc "25:00"), việc dùng ComboBox buộc người dùng chỉ được chọn các giá trị hợp lệ (0-23 cho giờ, 0-59 cho phút). Đây là một kỹ thuật **Phòng chống lỗi (Error Prevention)** ngay từ lớp giao diện, giảm thiểu đáng kể khối lượng code validate ở backend.
- **Tổ chức Layout (Hierarchy):** Cấu trúc lồng nhau: VBox (Form) -> HBox (Dòng thời gian) -> ComboBox (Giờ/Phút). Nhãn Label(":") được kẹp giữa hai ComboBox (dòng 1077) là một chi tiết UX nhỏ nhưng tinh tế, mô phỏng hiển thị của đồng hồ điện tử, giúp người dùng nhận diện ngay đây là trường nhập thời gian.
- **Trạng thái dữ liệu:** txtShiftId.setEditable(false): Khóa trường ID là chuẩn mực thiết kế cho các hệ thống sử dụng khóa chính tự tăng (Auto-increment Primary Key). Người dùng thấy ID để tham chiếu nhưng không thể can thiệp làm hỏng dữ liệu hệ thống.

b) Tạo Buttons và Event Handlers

```
    // Buttons
    HBox buttonBox = new HBox(10);
    buttonBox.setAlignment(Pos.CENTER);

    Button addButton = new Button("Add");
    addButton.setPrefWidth(80);
```

```
    addButton.setStyle("-fx-background-color:   #4CAF50;   -fx-
text-fill: white;");
    addButton.setOnAction(e -> {
        shiftController.handleAdd();
        if (calendarViewContainer != null) {
            refreshCalendarView(calendarViewContainer);
        }
    });

    Button updateButton = new Button("Update");
    updateButton.setPrefWidth(80);
    updateButton.setStyle("-fx-background-color: #2196F3; -fx-
text-fill: white;");
    updateButton.setOnAction(e -> {
        shiftController.handleUpdate();
        if (calendarViewContainer != null) {
            refreshCalendarView(calendarViewContainer);
        }
    });

    Button deleteButton = new Button("Delete");
    deleteButton.setPrefWidth(80);
    deleteButton.setStyle("-fx-background-color: #f44336; -fx-
text-fill: white;");
    deleteButton.setOnAction(e -> {
        shiftController.handleDelete();
        if (calendarViewContainer != null) {
            refreshCalendarView(calendarViewContainer);
        }
    });

    Button clearButton = new Button("Clear");
    clearButton.setPrefWidth(80);
    clearButton.setOnAction(e -> {
        shiftIdField.clear();
        employeeComboBox.setValue(null);
        datePicker.setValue(LocalDate.now());
        startHourComboBox.setValue(9);
        startMinuteComboBox.setValue(0);
        endHourComboBox.setValue(17);
        endMinuteComboBox.setValue(0);
        shiftTypeComboBox.setValue(null);
        if (shiftTable != null) {
            shiftTable.getSelectionModel().clearSelection();
        }
    });
```

Đoạn mã từ dòng 1098 đến 1145 thiết lập **Trung tâm Chỉ huy (Command Center)** của module Phân ca. Đây là nơi ý định của người dùng (muốn thêm, sửa, xóa) được chuyển đổi thành hành động cụ thể.

- **Semantics & UX (Ngữ nghĩa và Trải nghiệm người dùng):**
  - **Mã hóa màu sắc (Color Coding):** Việc sử dụng màu xanh lá (Add), xanh dương (Update) và đỏ (Delete) không chỉ là trang trí. Đây là tiêu chuẩn **Semantic UI**. Nó tận dụng tâm lý học màu sắc để giúp người dùng nhận diện nhanh hành động, giảm tải nhận thức (cognitive load) và hạn chế bấm nhầm nút nguy hiểm (như Xóa).
  - **Smart Defaults (Giá trị mặc định thông minh):** Trong nút Clear (dòng 1131), việc thiết lập giờ về "09:00 17:00" thay vì để trống hoặc 00:00 cho thấy tư duy hướng về quy trình nghiệp vụ. Đa số ca làm việc là giờ hành chính, nên thiết lập này giúp người dùng tiết kiệm được 4 thao tác chọn giờ cho mỗi lần nhập liệu.
- **Quy trình xử lý sự kiện (Event Handling Cycle):** Mỗi nút (Add/Update/Delete) tuân theo một mẫu thiết kế chặt chẽ gồm 3 bước:
  - **Trigger:** Người dùng bấm nút.
  - **Delegate:** View gọi Controller (shiftController.handleAdd()) để thực hiện logic nghiệp vụ và tương tác Database.
  - **Refresh:** View tự làm mới mình (refreshCalendarView()) để hiển thị trạng thái dữ liệu mới nhất.
- **Quản lý trạng thái Form (Form State Management):** Nút Clear (dòng 1144) thực hiện một nhiệm vụ quan trọng là shiftTable.getSelectionModel().clearSelection(). Dòng lệnh này chuyển ứng dụng từ **"Chế độ Chỉnh sửa"** (Edit Mode khi đang chọn một dòng trong bảng) về **"Chế độ Tạo mới"** (Create Mode). Nếu thiếu dòng này, người dùng có thể vô tình ghi đè lên ca làm việc cũ khi họ thực ra đang muốn tạo ca mới.

c) Đăng ký Form Fields và Hoàn thiện Form

```
    buttonBox.getChildren().addAll(addButton,  updateButton,
deleteButton, clearButton);
```

```
        // Set form fields in controller
        shiftController.setFormFields(shiftIdField,
employeeComboBox, datePicker,
                                      startHourComboBox,
startMinuteComboBox,
                                      endHourComboBox,
endMinuteComboBox, shiftTypeComboBox);

        formBox.getChildren().addAll(
                formLabel,
                new Label("ID:"),
                shiftIdField,
                new Label("Employee:"),
                employeeComboBox,
                new Label("Date:"),
                datePicker,
                startTimeBox,
                endTimeBox,
                new Label("Shift Type:"),
                shiftTypeComboBox,
                buttonBox
        );

        return formBox;
        }
```

Đoạn mã từ dòng 1147 đến 1169 thực hiện bước **lắp ráp cuối cùng (Final Assembly)** và thiết lập cơ chế giao tiếp sống còn giữa Giao diện (View) và Logic (Controller).

Điểm nhấn kiến trúc quan trọng nhất nằm ở dòng **1149-1152**, nơi phương thức shiftController.setFormFields(...) được kích hoạt. Đây thực chất là hành động **"trao quyền kiểm soát"**: MainApp truyền tham chiếu của 9 thành phần nhập liệu (TextField, ComboBox, DatePicker...) sang cho ShiftController. Nhờ cơ chế này, Controller thoát khỏi sự cô lập và có khả năng tương tác trực tiếp với giao diện: nó có thể **đọc dữ liệu** khi người dùng nhấn nút "Add", **ghi đè dữ liệu** vào form khi người dùng chọn một dòng trong bảng (Binding), và chủ động **nạp dữ liệu nguồn** (populate) cho các ComboBox như danh sách nhân viên hay loại ca.

Quy trình kết thúc ở dòng **1154-1167** với việc xếp chồng các thành phần vào container chính (formBox) theo một trật tự thị giác logic (Logical Visual Flow): đi từ

Tiêu đề xác định ngữ cảnh → Các trường thông tin chi tiết → Bộ chọn thời gian → và chốt lại bằng nhóm Nút hành động. Dòng **1169** trả về đối tượng form hoàn chỉnh, đánh dấu sự kết thúc của lớp Giao diện (View Layer) trong ứng dụng.

3.2.13 Phần Search và Filter Inventory: Method createInventorySearchSection()

Method này tạo phần tìm kiếm và lọc cho tab Inventory, cho phép người dùng tìm kiếm theo tên/ID và lọc theo danh mục và trạng thái tồn kho thấp.

a) Tạo Search Section

```
    /**
     * Create inventory search and filter section.
     */
    private VBox createInventorySearchSection() {
        VBox searchBox = new VBox(10);
        searchBox.setPadding(new Insets(10));
        searchBox.setStyle("-fx-background-color: white; -fx-
border-color: #ddd; -fx-border-radius: 5;");

        Label searchLabel = new Label("Search & Filter");
        searchLabel.setStyle("-fx-font-size: 16px; -fx-font-
weight: bold;");

        HBox searchRow = new HBox(10);
        searchRow.setAlignment(Pos.CENTER_LEFT);

        Label nameLabel = new Label("Search:");
        inventorySearchField = new TextField();
        inventorySearchField.setPromptText("Search by name or
ID...");
        inventorySearchField.setPrefWidth(200);

        Label categoryLabel = new Label("Category:");
        inventoryCategoryFilterComboBox = new ComboBox<>();
        inventoryCategoryFilterComboBox.getItems().addAll("All",
"Food", "Beverage", "Ingredient", "Cleaning", "Others");
        inventoryCategoryFilterComboBox.setValue("All");
        inventoryCategoryFilterComboBox.setPrefWidth(120);

        lowStockCheckBox = new CheckBox("Low Stock Only");

        Button searchButton = new Button("Apply Filter");
        searchButton.setStyle("-fx-background-color: #2196F3; -
fx-text-fill: white;");
```

```
        searchButton.setOnAction(e -> applyInventoryFilter());

        searchRow.getChildren().addAll(nameLabel,
inventorySearchField, categoryLabel,
                                       inventoryCategoryFilterComb
oBox, lowStockCheckBox, searchButton);

        searchBox.getChildren().addAll(searchLabel, searchRow);
        return searchBox;
    }
```

Đoạn mã từ dòng 1176 đến 1208 chịu trách nhiệm xây dựng **Thanh công cụ Tìm kiếm và Lọc (Search & Filter Toolbar)** cho phân hệ Kho hàng (Inventory). Về mặt cấu trúc, phương thức này sử dụng VBox làm container chính để duy trì sự nhất quán với giao diện tổng thể, bên trong lồng ghép một HBox (dòng 1184) để dàn trải các công cụ điều khiển theo phương ngang, giúp tối ưu hóa không gian màn hình.

Hệ thống cung cấp một bộ công cụ lọc đa chiều bao gồm: TextField có chiều rộng 200px với hướng dẫn tìm kiếm linh hoạt (theo Tên hoặc ID), ComboBox phân loại danh mục sản phẩm (với trạng thái mặc định an toàn là "All"), và đặc biệt là CheckBox "Low Stock" (dòng 1198) một tính năng nghiệp vụ quan trọng giúp người quản lý nhanh chóng tách lọc các mặt hàng sắp hết. Quy trình tương tác được hoàn tất bởi nút "Apply Filter" với định dạng màu xanh dương (Action Color), đóng vai trò kích hoạt phương thức xử lý logic applyInventoryFilter(). Khối mã kết thúc bằng việc lắp ráp các thành phần vào container và trả về, tạo nên một giao diện truy xuất dữ liệu trực quan và hiệu quả.

b) Method applyInventoryFilter()

```
    /**
     * Apply inventory filter.
     */
    private void applyInventoryFilter() {
        String searchText = inventorySearchField.getText();
        String                  categoryFilter                  =
inventoryCategoryFilterComboBox.getValue();
        if ("All".equals(categoryFilter)) {
            categoryFilter = null;
        }
        boolean lowStockOnly = lowStockCheckBox.isSelected();
```

```
        javafx.collections.transformation.FilteredList<InventoryI
tem> filtered =
                inventoryController.filterItems(searchText,
categoryFilter, lowStockOnly);
        inventoryTable.setItems(filtered);
    }
```

Đoạn mã từ dòng 1214 đến 1224 thực thi logic kết nối giữa bộ điều khiển giao diện và công cụ xử lý dữ liệu. Khi hành động lọc được kích hoạt, phương thức này đóng vai trò là một **Bộ thu thập thông tin (Information Aggregator)**: nó trích xuất đồng thời từ khóa tìm kiếm, trạng thái checkbox "Low Stock", và danh mục sản phẩm. Một điểm sáng về kỹ thuật lập trình phòng thủ (defensive programming) xuất hiện ở dòng 1217-1219: việc chuẩn hóa giá trị "All" thành null thông qua phép so sánh hằng số ("All".equals(...)) giúp loại bỏ hoàn toàn rủi ro NullPointerException và đơn giản hóa logic kiểm tra ở phía backend.

Thay vì tự xử lý thuật toán tìm kiếm phức tạp, MainApp tuân thủ triệt để mẫu thiết kế **Ủy quyền (Delegation)** bằng cách chuyển giao toàn bộ tham số cho inventoryController.filterItems(). Kết quả trả về là một FilteredList—một cấu trúc dữ liệu mạnh mẽ của JavaFX có khả năng quan sát và tự động cập nhật—được nạp trực tiếp vào inventoryTable, đảm bảo bảng dữ liệu phản hồi tức thì và chính xác theo tiêu chí người dùng vừa nhập.

3.2.14 Phần Bảng Inventory: Method createInventoryTableSection()

Method này tạo bảng hiển thị danh sách inventory items với các cột thông tin chi tiết và cột status hiển thị cảnh báo tồn kho thấp.

a) Tạo bảng và cột cơ bản

```
    /**
     * Create inventory table section.
     */
    private VBox createInventoryTableSection() {
        VBox tableBox = new VBox(10);
        tableBox.setPadding(new Insets(10));
        tableBox.setStyle("-fx-background-color:    white;    -fx-
border-color: #ddd; -fx-border-radius: 5;");

        Label tableLabel = new Label("Inventory Items");
```

```
        tableLabel.setStyle("-fx-font-size:    16px;    -fx-font-
weight: bold;");

        inventoryTable = new TableView<>();
        inventoryTable.setColumnResizePolicy(TableView.CONSTRAINE
D_RESIZE_POLICY);
        inventoryTable.setPrefHeight(600);

        // Table columns
        TableColumn<InventoryItem,   String>   idColumn   =   new
TableColumn<>("ID");
        idColumn.setCellValueFactory(new
PropertyValueFactory<>("id"));
        idColumn.setPrefWidth(80);

        TableColumn<InventoryItem,   String>   nameColumn   =   new
TableColumn<>("Name");
        nameColumn.setCellValueFactory(new
PropertyValueFactory<>("name"));
        nameColumn.setPrefWidth(150);

        TableColumn<InventoryItem, String> categoryColumn = new
TableColumn<>("Category");
        categoryColumn.setCellValueFactory(new
PropertyValueFactory<>("category"));
        categoryColumn.setPrefWidth(100);

        TableColumn<InventoryItem, Double> quantityColumn = new
TableColumn<>("Quantity");
        quantityColumn.setCellValueFactory(new
PropertyValueFactory<>("quantity"));
        quantityColumn.setPrefWidth(80);

        TableColumn<InventoryItem,   String>   unitColumn   =   new
TableColumn<>("Unit");
        unitColumn.setCellValueFactory(new
PropertyValueFactory<>("unit"));
        unitColumn.setPrefWidth(60);

        TableColumn<InventoryItem, Double> thresholdColumn = new
TableColumn<>("Min Threshold");
        thresholdColumn.setCellValueFactory(new
PropertyValueFactory<>("minimumThreshold"));
        thresholdColumn.setPrefWidth(100);

        TableColumn<InventoryItem, String> supplierColumn = new
TableColumn<>("Supplier");
        supplierColumn.setCellValueFactory(new
PropertyValueFactory<>("supplierName"));
```

```
        supplierColumn.setPrefWidth(120);

        TableColumn<InventoryItem, String> locationColumn = new
TableColumn<>("Location");
        locationColumn.setCellValueFactory(new
PropertyValueFactory<>("storageLocation"));
        locationColumn.setPrefWidth(100);
```

Đoạn mã từ dòng 1230 đến 1273 chịu trách nhiệm xây dựng **trung tâm hiển thị dữ liệu kho hàng (Inventory Dashboard)**. Cấu trúc giao diện bắt đầu với việc khởi tạo container VBox và tiêu đề, tuân thủ nghiêm ngặt chuẩn mực thiết kế thống nhất của ứng dụng.

Trọng tâm kỹ thuật nằm ở việc cấu hình TableView (dòng 1238-1240). Việc áp dụng chính sách CONSTRAINED_RESIZE_POLICY là một quyết định UX quan trọng, đảm bảo bảng dữ liệu luôn tận dụng tối đa chiều rộng màn hình, loại bỏ các khoảng trắng thừa thãi hoặc thanh cuộn ngang không cần thiết.

Hệ thống cột (dòng 1243-1273) được thiết kế bao quát toàn bộ vòng đời quản lý hàng tồn kho: từ định danh cơ bản (ID, Name, Category) đến các chỉ số vận hành quan trọng như Số lượng (Quantity), Đơn vị (Unit), và đặc biệt là Ngưỡng tối thiểu (Min Threshold) để kích hoạt cảnh báo. Cơ chế liên kết dữ liệu (Data Binding) thông qua PropertyValueFactory đảm bảo sự đồng bộ tự động giữa đối tượng model InventoryItem và giao diện, giúp bảng hiển thị chính xác mọi thay đổi về vị trí (Location) hay nhà cung cấp (Supplier) mà không cần can thiệp thủ công.

b) Cột Status với Lambda Expression

```
    // Add low stock indicator column
    TableColumn<InventoryItem, String> statusColumn = new
TableColumn<>("Status");
    statusColumn.setCellValueFactory(cellData -> {
        InventoryItem item = cellData.getValue();
        String status = item.isLowStock() ? "⚠ Low Stock" : "✓
OK";
        return new
javafx.beans.property.SimpleStringProperty(status);
    });
    statusColumn.setPrefWidth(100);

    inventoryTable.getColumns().addAll(idColumn, nameColumn,
categoryColumn, quantityColumn,
```

```
                                        unitColumn,
thresholdColumn, supplierColumn, locationColumn, statusColumn);
```

Đoạn mã từ dòng 1275 đến 1285 triển khai cột trạng thái ("Status") với một cách tiếp cận thông minh về hiển thị dữ liệu. Khác với các cột trước đó chỉ đơn thuần liên kết thuộc tính, cột này sử dụng **Biểu thức Lambda** (dòng 1277-1281) để thực hiện **tính toán giá trị động (Computed Value)** ngay tại thời điểm render. Thay vì hiển thị con số khô khan, logic cellValueFactory chủ động gọi phương thức nghiệp vụ isLowStock() để đối chiếu lượng tồn kho với ngưỡng tối thiểu. Kết quả được trực quan hóa ngay lập tức thông qua các ký tự biểu tượng: cảnh báo "⚠ Low Stock" gây sự chú ý khi hàng thiếu, hoặc "✓ OK" để xác nhận an toàn. Cách thiết kế này giúp giảm tải nhận thức cho người quản lý, cho phép họ nhận diện vấn đề chỉ qua một cái liếc mắt mà không cần tự thực hiện phép so sánh toán học. Cuối cùng, dòng 1284-1285 hoàn tất cấu trúc bảng bằng việc tập hợp tất cả các cột dữ liệu và cột chức năng này vào TableView.

c) Ràng buộc Controller và Xử lý Selection

```
      // Bind table to controller
      inventoryController.setItemTableView(inventoryTable);
      inventoryTable.setItems(inventoryController.getItemList());

      // Handle row selection
      inventoryTable.getSelectionModel().selectedItemProperty().add
Listener(
                (obs, oldSelection, newSelection) -> {
                    if (newSelection != null) {
                        inventoryController.handleTableSelection();
                    }
                }
      );

      tableBox.getChildren().addAll(tableLabel, inventoryTable);
      return tableBox;
      }
```

Đoạn mã từ dòng 1288 đến 1301 hoàn tất việc thiết lập cơ chế **liên kết dữ liệu (Data Binding)** và tương tác cho bảng kho hàng. Bằng việc đăng ký inventoryTable với Controller và gán nguồn dữ liệu thông qua phương thức setItems (dòng 1288-1289), ứng dụng thiết lập một luồng đồng bộ hóa tự động: mọi thay đổi trong danh

sách ObservableList của Controller sẽ lập tức phản ánh lên giao diện mà không cần can thiệp thủ công.

Bên cạnh đó, logic xử lý tương tác người dùng tuân thủ triệt để mẫu thiết kế **Ủy quyền (Delegation)**. Thay vì xử lý trực tiếp tại View, sự kiện chọn hàng (Selection Listener, dòng 1292-1298) được chuyển tiếp ngay lập tức sang phương thức handleTableSelection của Controller để điền dữ liệu vào Form, đảm bảo sự phân tách rõ ràng giữa giao diện hiển thị và logic nghiệp vụ. Khối mã kết thúc bằng việc đóng gói các thành phần vào container, sẵn sàng hiển thị.

3.2.15 Phần Form Inventory: Method createInventoryFormSection()

Method này tạo phần form cho inventory, bao gồm nhiều form con được bọc trong ScrollPane để có thể cuộn khi nội dung dài.

```
      /**
       * Create inventory form section (Item form + Stock form)
wrapped in ScrollPane.
       */
      private ScrollPane createInventoryFormSection() {
          VBox formSection = new VBox(15);
          formSection.setPadding(new Insets(10));

          // Item Management Form
          VBox itemFormBox = createInventoryItemForm();

          // Stock In/Out Form
          VBox stockFormBox = createStockInOutForm();

          // Transaction History
          VBox transactionBox = createTransactionHistory();

          formSection.getChildren().addAll(itemFormBox,
stockFormBox, transactionBox);

          // Wrap in ScrollPane to allow scrolling when content
exceeds available space
          ScrollPane scrollPane = new ScrollPane(formSection);
          scrollPane.setFitToWidth(true);
          scrollPane.setHbarPolicy(ScrollPane.ScrollBarPolicy.NEVER
);
          scrollPane.setVbarPolicy(ScrollPane.ScrollBarPolicy.AS_NE
EDED);
          scrollPane.setStyle("-fx-background: white;");
```

```
        return scrollPane;
    }
```

Đoạn mã từ dòng 1307 đến 1329 chịu trách nhiệm kiến tạo **khu vực quản lý tác vụ (Task Management Area)** cho phân hệ Kho hàng. Thay vì giới hạn nội dung trong một khung nhìn tĩnh, phương thức trả về một ScrollPane (dòng 1307) một quyết định UX thiết yếu để chứa đựng khối lượng chức năng lớn gồm ba module riêng biệt được xếp chồng lên nhau: Form quản lý sản phẩm, Form Nhập/Xuất kho và Bảng lịch sử giao dịch.

Cấu trúc mã tuân thủ triệt để nguyên tắc **Module hóa (Modularity)**: thay vì nhồi nhét code UI vào một chỗ, nó gọi các phương thức helper chuyên biệt (createInventoryItemForm, createStockInOutForm, v.v.) để lắp ráp giao diện. Điểm nhấn kỹ thuật nằm ở việc cấu hình ScrollPane (dòng 1323-1327): bằng cách kết hợp setFitToWidth(true) với việc vô hiệu hóa thanh cuộn ngang (HbarPolicy.NEVER), ứng dụng ép buộc các form con phải tự co giãn để lấp đầy chiều rộng khả dụng. Điều này tạo ra trải nghiệm người dùng liền mạch, loại bỏ thao tác cuộn ngang khó chịu và đảm bảo tất cả công cụ nghiệp vụ đều có thể truy cập được thông qua thao tác cuộn dọc tự nhiên.

3.2.16 Form Quản lý Item: Method createInventoryItemForm()

Method này tạo form để thêm, cập nhật và xóa inventory items.

a) Tạo Form Fields

```
    /**
     * Create inventory item form.
     */
    private VBox createInventoryItemForm() {
        VBox formBox = new VBox(10);
        formBox.setPadding(new Insets(10));
        formBox.setStyle("-fx-background-color:    white;    -fx-border-
color: #ddd; -fx-border-radius: 5;");

        Label formLabel = new Label("Item Management");
        formLabel.setStyle("-fx-font-size:    16px;    -fx-font-weight:
bold;");

        // Form fields
        inventoryIdField = new TextField();
        inventoryIdField.setPromptText("ID (auto-generated)");
```

```
        inventoryIdField.setEditable(false);
        inventoryIdField.setStyle("-fx-background-color: #f0f0f0;");

        inventoryNameField = new TextField();
        inventoryNameField.setPromptText("Item Name *");

        inventoryCategoryComboBox = new ComboBox<>();
        inventoryCategoryComboBox.setPromptText("Category *");

        inventoryUnitComboBox = new ComboBox<>();
        inventoryUnitComboBox.setPromptText("Unit *");

        inventoryQuantityField = new TextField();
        inventoryQuantityField.setPromptText("Quantity *");

        inventoryThresholdField = new TextField();
        inventoryThresholdField.setPromptText("Minimum Threshold *");

        inventorySupplierField = new TextField();
        inventorySupplierField.setPromptText("Supplier Name");

        inventoryStorageComboBox = new ComboBox<>();
        inventoryStorageComboBox.setPromptText("Storage Location *");
```

Đoạn mã từ dòng 1335 đến 1368 tập trung xây dựng **Giao diện Quản lý Chi tiết Hàng hóa (Item Management Form)**. Bắt đầu với một VBox được định dạng nhất quán, giao diện được thiết kế để định hướng hành vi nhập liệu của người dùng ngay từ cái nhìn đầu tiên.

Điểm đáng chú ý về UX là sự phân biệt rõ ràng giữa dữ liệu hệ thống và dữ liệu người dùng: trường ID (dòng 1344) bị khóa (read-only) và tô nền xám, ngăn chặn các can thiệp vô ý vào khóa chính của cơ sở dữ liệu. Đối với các thuộc tính phân loại như *Category*, *Unit* hay *Storage Location*, ứng dụng ưu tiên sử dụng ComboBox thay vì TextField. Chiến lược này không chỉ giúp **chuẩn hóa dữ liệu đầu vào** (tránh lỗi chính tả như "kg" vs "Kg") mà còn tạo điểm neo để Controller nạp danh sách tùy chọn chuẩn sau này. Cuối cùng, việc gán tất cả các thành phần giao diện này vào các biến instance (field instance) là bước chuẩn bị thiết yếu, mở đường cho Controller truy xuất và thao tác dữ liệu trong các sự kiện lưu hoặc chỉnh sửa sau này.

b) Tạo Buttons và Hoàn thiện Form

```
    // Buttons
    HBox buttonBox = new HBox(10);
    buttonBox.setAlignment(Pos.CENTER);
```

```
     Button addButton = new Button("Add");
     addButton.setPrefWidth(70);
     addButton.setStyle("-fx-background-color: #4CAF50; -fx-text-
fill: white;");
     addButton.setOnAction(e -> inventoryController.handleAdd());

     Button updateButton = new Button("Update");
     updateButton.setPrefWidth(70);
     updateButton.setStyle("-fx-background-color: #2196F3; -fx-
text-fill: white;");
     updateButton.setOnAction(e                                  ->
inventoryController.handleUpdate());

     Button deleteButton = new Button("Delete");
     deleteButton.setPrefWidth(70);
     deleteButton.setStyle("-fx-background-color: #f44336; -fx-
text-fill: white;");
     deleteButton.setOnAction(e                                  ->
inventoryController.handleDelete());

     Button clearButton = new Button("Clear");
     clearButton.setPrefWidth(70);
     clearButton.setOnAction(e -> {
         inventoryIdField.clear();
         inventoryNameField.clear();
         inventoryCategoryComboBox.setValue(null);
         inventoryUnitComboBox.setValue(null);
         inventoryQuantityField.clear();
         inventoryThresholdField.clear();
         inventorySupplierField.clear();
         inventoryStorageComboBox.setValue(null);
         inventoryTable.getSelectionModel().clearSelection();
     });

     buttonBox.getChildren().addAll(addButton,       updateButton,
deleteButton, clearButton);

     // Set form fields in controller
     inventoryController.setItemFormFields(inventoryIdField,
inventoryNameField, inventoryCategoryComboBox,
                                           inventoryUnitComboBox,
inventoryQuantityField,
                                           inventoryThresholdField,
inventorySupplierField,
                                           inventoryStorageComboBox
);

     formBox.getChildren().addAll(
```

```
            formLabel,
            new Label("ID:"),
            inventoryIdField,
            new Label("Name:"),
            inventoryNameField,
            new Label("Category:"),
            inventoryCategoryComboBox,
            new Label("Unit:"),
            inventoryUnitComboBox,
            new Label("Quantity:"),
            inventoryQuantityField,
            new Label("Min Threshold:"),
            inventoryThresholdField,
            new Label("Supplier:"),
            inventorySupplierField,
            new Label("Storage Location:"),
            inventoryStorageComboBox,
            buttonBox
    );

    return formBox;
    }
```

Đoạn mã từ dòng 1371 đến 1432 hoàn tất việc xây dựng **cơ chế điều khiển và kết nối (Control & Connectivity Mechanism)** cho form kho hàng. Hệ thống các nút chức năng (Add, Update, Delete) được thiết lập tuân thủ nghiêm ngặt mô hình **Ủy quyền (Delegation)**: giao diện chỉ đóng vai trò bộ kích hoạt (trigger), còn toàn bộ gánh nặng xử lý logic nghiệp vụ được chuyển tiếp ngay lập tức sang InventoryController. Riêng nút "Clear" đảm nhận vai trò quản lý trạng thái giao diện, giúp người dùng thoát khỏi ngữ cảnh "Chỉnh sửa" để quay về trạng thái "Tạo mới" sạch sẽ.

Điểm trọng yếu về mặt kiến trúc nằm ở dòng 1405-1409 với thao tác **"đấu nối" (Wiring)**. Thông qua phương thức setFormFields, MainApp chính thức trao quyền kiểm soát toàn bộ các ô nhập liệu cho Controller. Nhờ "cái bắt tay" này, Controller thoát khỏi sự cô lập: nó có thể đọc dữ liệu đầu vào, tự động điền thông tin khi người dùng chọn hàng trong bảng, và chủ động nạp dữ liệu nguồn cho các ComboBox (như danh sách Nhà cung cấp hay Đơn vị tính). Cuối cùng, các thành phần được lắp ráp theo trật tự logic từ trên xuống dưới, tạo nên một giao diện CRUD hoàn chỉnh và chặt chẽ.

3.2.17 Form Stock In/Out: Method createStockInOutForm()

Method này tạo form để thực hiện nhập kho (stock in) và xuất kho (stock out), tạo transaction records.

```
    /**
     * Create stock in/out form.
     */
    private VBox createStockInOutForm() {
        VBox formBox = new VBox(10);
        formBox.setPadding(new Insets(10));
        formBox.setStyle("-fx-background-color:    white;    -fx-
border-color: #ddd; -fx-border-radius: 5;");

        Label formLabel = new Label("Stock In/Out");
        formLabel.setStyle("-fx-font-size: 16px; -fx-font-weight:
bold;");

        stockItemComboBox = new ComboBox<>();
        stockItemComboBox.setPromptText("Select Item *");
        stockItemComboBox.setPrefWidth(Double.MAX_VALUE);

        stockTypeComboBox = new ComboBox<>();
        stockTypeComboBox.setPromptText("Type *");
        stockTypeComboBox.setPrefWidth(Double.MAX_VALUE);

        stockQuantityField = new TextField();
        stockQuantityField.setPromptText("Quantity *");

        stockReasonComboBox = new ComboBox<>();
        stockReasonComboBox.setPromptText("Reason *");
        stockReasonComboBox.setPrefWidth(Double.MAX_VALUE);

        Button stockButton = new Button("Process Stock");
        stockButton.setPrefWidth(Double.MAX_VALUE);
        stockButton.setStyle("-fx-background-color: #FF9800; -fx-
text-fill: white;");
        stockButton.setOnAction(e -> {
            // Use first employee as default staff (in real app,
get from session)
            String staffId = "SYS";
            String staffName = "System";
            inventoryController.handleStockOperation(staffId,
staffName);
        });

        // Set form fields in controller
```

```
        inventoryController.setStockFormFields(stockItemComboBox,
stockQuantityField,
                                               stockReasonComboBox
, stockTypeComboBox);

        formBox.getChildren().addAll(
                formLabel,
                new Label("Item:"),
                stockItemComboBox,
                new Label("Type:"),
                stockTypeComboBox,
                new Label("Quantity:"),
                stockQuantityField,
                new Label("Reason:"),
                stockReasonComboBox,
                stockButton
        );

        return formBox;
    }
```

Đoạn mã từ dòng 1438 đến 1486 tập trung xây dựng giao diện cho **nghiệp vụ kho vận cốt lõi (Core Inventory Operations)**: Nhập và Xuất kho. Khác với form CRUD quản lý thông tin sản phẩm tĩnh ở trên, form này mang tính chất **giao dịch (Transactional)**, nơi dữ liệu biến đổi liên tục theo thời gian thực.

Chiến lược thiết kế UI ưu tiên sự an toàn và chính xác của dữ liệu: 3 trong 4 trường nhập liệu (Sản phẩm, Loại giao dịch, Lý do) sử dụng ComboBox. Điều này ép buộc người dùng chọn từ các giá trị chuẩn hóa do Controller cung cấp, loại bỏ hoàn toàn rủi ro lỗi nhập liệu thủ công (như gõ sai tên sản phẩm hay lý do không hợp lệ).

Điểm nhấn thị giác nằm ở nút "Process Stock" (dòng 1461) với tông màu cam (#FF9800). Sự lựa chọn màu sắc này không chỉ mang tính thẩm mỹ mà còn là tín hiệu ngữ nghĩa (Semantic Signal), giúp người dùng phân biệt rõ ràng giữa hành động "Xử lý dòng hàng" với các thao tác quản lý dữ liệu thông thường (Thêm/Sửa/Xóa). Về mặt logic, nút này kích hoạt phương thức handleStockOperation trong Controller, đóng vai trò cầu nối để thực hiện chuỗi hành động kép: vừa cập nhật số lượng tồn kho trong Database, vừa ghi lại dấu vết vào lịch sử giao dịch (Transaction Log).

3.2.18 Transaction History: Method createTransactionHistory()

Method này tạo bảng hiển thị lịch sử các giao dịch nhập/xuất kho gần đây.

```
    /**
     * Create transaction history section.
     */
    private VBox createTransactionHistory() {
        VBox transactionBox = new VBox(10);
        transactionBox.setPadding(new Insets(10));
        transactionBox.setStyle("-fx-background-color:  white;  -
fx-border-color: #ddd; -fx-border-radius: 5;");
        transactionBox.setPrefHeight(250);

        Label transactionLabel = new Label("Recent Transactions");
        transactionLabel.setStyle("-fx-font-size: 14px; -fx-font-
weight: bold;");

        transactionTable = new TableView<>();
        transactionTable.setColumnResizePolicy(TableView.CONSTRAI
NED_RESIZE_POLICY);
        transactionTable.setPrefHeight(200);

        // Table columns
        TableColumn<InventoryTransaction, String> itemColumn = new
TableColumn<>("Item");
        itemColumn.setCellValueFactory(new
PropertyValueFactory<>("itemName"));
        itemColumn.setPrefWidth(120);

        TableColumn<InventoryTransaction, Double> qtyColumn = new
TableColumn<>("Qty");
        qtyColumn.setCellValueFactory(new
PropertyValueFactory<>("quantity"));
        qtyColumn.setPrefWidth(60);

        TableColumn<InventoryTransaction, String> typeColumn = new
TableColumn<>("Type");
        typeColumn.setCellValueFactory(new
PropertyValueFactory<>("type"));
        typeColumn.setPrefWidth(50);

        TableColumn<InventoryTransaction, String> reasonColumn =
new TableColumn<>("Reason");
        reasonColumn.setCellValueFactory(new
PropertyValueFactory<>("reason"));
        reasonColumn.setPrefWidth(80);
```

```
        TableColumn<InventoryTransaction, String> staffColumn =
new TableColumn<>("Staff");
        staffColumn.setCellValueFactory(new
PropertyValueFactory<>("staffName"));
        staffColumn.setPrefWidth(80);

        TableColumn<InventoryTransaction, String> timeColumn = new
TableColumn<>("Time");
        timeColumn.setCellValueFactory(cellData -> {
            var time = cellData.getValue().getTimestamp();
            return                                          new
javafx.beans.property.SimpleStringProperty(
                time            !=            null            ?
time.format(DateTimeFormatter.ofPattern("MM/dd HH:mm")) : ""
            );
        });
        timeColumn.setPrefWidth(100);

        transactionTable.getColumns().addAll(itemColumn,
qtyColumn, typeColumn, reasonColumn, staffColumn, timeColumn);

        // Bind table to controller
        inventoryController.setTransactionTableView(transactionTa
ble);
        transactionTable.setItems(inventoryController.getTransact
ionList());

        transactionBox.getChildren().addAll(transactionLabel,
transactionTable);
        return transactionBox;
    }
```

Đoạn mã từ dòng 1494 đến 1544 hoàn thiện giao diện kho hàng bằng việc xây dựng thành phần **Lịch sử Giao dịch (Transaction History)**. Đây là một thành phần phụ trợ (Secondary Component) nhưng có vai trò chiến lược trong việc **Kiểm toán (Auditing)**: nó ghi lại dấu vết của mọi biến động kho hàng. Về mặt thiết kế, khu vực này được giới hạn chiều cao cứng ở mức 250px và tiêu đề nhỏ hơn (14px), đảm bảo cung cấp thông tin minh bạch mà không tranh chấp không gian với các form nghiệp vụ chính phía trên.

Điểm sáng kỹ thuật nằm ở cột thời gian ("Time"). Thay vì hiển thị chuỗi ngày giờ thô kệch mặc định, ứng dụng sử dụng **Lambda Expression** (dòng 1523-1525) để định dạng lại đối tượng LocalDateTime thành chuỗi ngắn gọn "MM/dd HH:mm". Đây là một xử lý UX tinh tế giúp tiết kiệm không gian ngang quý giá của bảng. Cuối

cùng, việc liên kết dữ liệu trực tiếp với Controller biến bảng này thành một màn hình giám sát thời gian thực: ngay khi nút "Process Stock" ở trên được nhấn, dòng giao dịch mới sẽ lập tức xuất hiện ở đây, xác nhận hành động đã thành công.

3.2.19 Tổng kết

Hiện thực class MainApp thể hiện một số pattern nhất quán phù hợp với thiết kế kiến trúc phân lớp:

- **Pattern Dependency Injection**: Service được tạo trong MainApp và được inject vào controller thông qua constructor, cho MainApp quyền kiểm soát vòng đời và chia sẻ phụ thuộc.
- **Tạo UI Phân cấp**: UI được xây dựng theo các lớp: main layout → tab pane → tabs → sections → thành phần riêng lẻ. Mỗi cấp được tạo bởi các method chuyên dụng, thúc đẩy tổ chức và khả năng bảo trì.
- **Pattern Ủy quyền Sự kiện**: Sự kiện UI được bắt trong MainApp, nhưng thực thi logic nghiệp vụ được ủy quyền cho controller. Điều này duy trì tách biệt mối quan tâm trong khi cho phép MainApp điều phối cập nhật UI.
- **Pattern Ràng buộc Rõ ràng**: Thành phần UI được đăng ký rõ ràng với controller thông qua lời gọi method (setTableView(), setFormFields()). Điều này làm cho các kết nối rõ ràng và cho phép controller quản lý luồng dữ liệu.
- **Điều phối Giữa các Controller**: MainApp điều phối giữa các controller khi thay đổi UI trong một tab ảnh hưởng đến tab khác (ví dụ: thêm nhân viên làm mới dropdown nhân viên của tab shift).

Các pattern này cùng nhau thể hiện cách class MainApp hoàn thành vai trò của nó như người điều phối Tầng Application: nó khởi tạo phụ thuộc, tạo và điều phối thành phần UI, quản lý ràng buộc sự kiện, và ủy quyền logic nghiệp vụ cho controller phù hợp, tất cả trong khi duy trì các ranh giới kiến trúc được định nghĩa trong thiết kế UML.

Phần ba nói vê cách cấu trúc hiện thực một ứng dụng Java UML để hiện thực đề tài này nếu như chúng ta dành không gian để xây dựng một bộ nhớ dữ liệu hợp lý cho khung chương tình ới thi nhiều khả năng

3.3 Tầng Controller

Tầng Controller, được đại diện bởi các class controller trong biểu đồ UML, đóng vai trò là lớp trung gian giữa Tầng Application (UI) và Tầng Service (business logic). Trong kiến trúc phân lớp được định nghĩa ở Chương 2, tầng này chịu trách nhiệm xử lý tương tác người dùng, điều phối luồng dữ liệu giữa UI và Service, và quản lý trạng thái hiển thị. Phân tích chi tiết này xem xét từng thành phần của source code các controller với các đoạn code cụ thể và giải thích từng dòng để chứng minh cách thiết kế kiến trúc được chuyển đổi thành hiện thực thực tế.

3.3.1 Phân tích BookingController.java

'BookingController' là controller chịu trách nhiệm quản lý các tương tác liên quan đến đặt bàn (booking) trong hệ thống nhà hàng. Controller này xử lý các thao tác CRUD cơ bản (Create, Read, Update, Delete) cũng như các thao tác nghiệp vụ đặc biệt như hủy đặt bàn và xác nhận khách đã đến.

3.3.1.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
import com.restaurantmanagement.model.Booking;
import com.restaurantmanagement.service.BookingService;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
import javafx.scene.control.DatePicker;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;

import java.time.LocalDate;
import java.time.LocalTime;
```

Đoạn mã từ dòng 1 đến 14 thiết lập nền tảng kiến trúc cho Controller, định vị nó rõ ràng trong package com.restaurantmanagement.controller với vai trò trung gian của tầng ứng dụng. Danh sách import thể hiện sự hội tụ của ba thành phần cốt lõi cần thiết cho một Controller hiện đại: **Logic nghiệp vụ** (thông qua Booking model và BookingService), **Cơ sở hạ tầng Giao diện** (với bộ công cụ JavaFX như ObservableList cho data binding và các controls nhập liệu), và **Xử lý dữ liệu** (với

java.time để quản lý lịch trình chính xác). Tổng thể, phần khai báo này phác họa chân dung của một lớp điều phối điển hình trong mô hình MVC, đóng vai trò cầu nối luân chuyển dữ liệu hai chiều giữa Service layer và Giao diện người dùng.

b) Khai báo Class và JavaDoc

```
/**
 * Controller for Booking management UI.
 * Handles user interactions and updates the view.
 */
public class BookingController {
```

Đoạn mã từ dòng 16 đến 20 xác định danh tính và phạm vi hoạt động của lớp điều khiển. Thông qua phần tài liệu JavaDoc (dòng 16-19), vai trò của class được xác định rõ ràng là **Bộ quản lý giao diện đặt bàn**, chịu trách nhiệm tiếp nhận tương tác từ người dùng và phản hồi lại bằng cách cập nhật hiển thị (View). Khai báo public ở dòng 20 mở rộng phạm vi truy cập của lớp, đảm bảo rằng các thành phần cốt lõi khác nằm khác package—đặc biệt là MainApp—có thể khởi tạo và tích hợp nó vào luồng chạy chính của ứng dụng.

c) Khai báo Field Service và Data

```
private final BookingService bookingService;
private final ObservableList<Booking> bookingList;
private TableView<Booking> tableView;
```

Đoạn mã từ dòng 21 đến 23 thiết lập nền tảng dữ liệu và cấu trúc phụ thuộc cho Controller. Đầu tiên, dòng 21 định nghĩa mối quan hệ cốt lõi với tầng dịch vụ thông qua BookingService; việc áp dụng từ khóa final tại đây đảm bảo tính bất biến (immutability), khẳng định rằng logic nghiệp vụ là cố định và an toàn suốt vòng đời của đối tượng.

Tiếp theo, dòng 22 khởi tạo ObservableList, đóng vai trò là cầu nối dữ liệu "sống" giữa logic và giao diện: nhờ cơ chế quan sát (Observer pattern) của JavaFX, mọi biến động trong danh sách này sẽ tự động được phản ánh lên UI mà không cần can thiệp thủ công. Cuối cùng, dòng 23 giữ tham chiếu đến TableView (được tiêm vào sau qua setter), cho phép Controller nắm quyền kiểm soát trực tiếp đối với thành phần hiển thị, đặc biệt là trong việc xử lý các tương tác chọn dòng (selection model) của người dùng.

d) Khai báo Field UI Components

```
    // UI Components (for form inputs)
    private TextField idField;
    private TextField customerNameField;
    private TextField phoneNumberField;
    private TextField numberOfGuestsField;
    private DatePicker datePicker;
    private ComboBox<Integer> hourComboBox;
    private ComboBox<Integer> minuteComboBox;
    private TextField tableIdField;
    private ComboBox<String> statusComboBox;
```

Đoạn mã từ dòng 25 đến 34 định nghĩa tập hợp các điều khiển giao diện (UI Controls) đóng vai trò là "cổng nhập liệu" cho quy trình đặt bàn. Các trường thông tin cơ bản như tên khách, số điện thoại và số lượng khách được thu thập qua TextField tiêu chuẩn, trong khi idField thường mang tính chất hiển thị (read-only) để định danh bản ghi hệ thống.

Điểm nhấn trong thiết kế trải nghiệm người dùng (UX) nằm ở cơ chế xử lý thời gian: thay vì để người dùng nhập văn bản tự do dễ gây lỗi định dạng, Controller sử dụng DatePicker kết hợp với hai ComboBox riêng biệt cho Giờ và Phút. Chiến lược này giúp chuẩn hóa dữ liệu đầu vào ngay từ nguồn, loại bỏ hoàn toàn các lỗi cú pháp thời gian phổ biến.

Về mặt kiến trúc, tất cả các biến này đều được khai báo private để đảm bảo tính đóng gói. Chúng ở trạng thái chờ (null) cho đến khi được MainApp "kết nối" (wire) thông qua phương thức setFormFields, thể hiện rõ cơ chế Tiêm phụ thuộc (Dependency Injection) thủ công mà ứng dụng đang áp dụng để tách biệt việc khởi tạo giao diện khỏi logic xử lý.

3.3.1.2 Constructor và khởi tạo

```
    ublic BookingController() {
        this.bookingService = new BookingService();
        this.bookingList = FXCollections.observableArrayList();
        loadBookings();
    }
```

Đoạn mã từ dòng 36 đến 39 mô tả quy trình khởi tạo (Constructor) của BookingController. Điểm đáng chú ý đầu tiên là chiến lược quản lý phụ thuộc: khác

với mô hình Dependency Injection thấy ở EmployeeController, controller này tự chủ động khởi tạo BookingService (dòng 37). Điều này cho thấy sự linh hoạt trong thiết kế kiến trúc, cho phép module đặt bàn hoạt động như một đơn vị độc lập tự quản lý vòng đời dịch vụ của mình.

Tiếp theo, hạ tầng dữ liệu được thiết lập thông qua việc tạo ObservableList rỗng bằng FXCollections (dòng 38), chuẩn bị sẵn sàng cho cơ chế ràng buộc dữ liệu tự động của JavaFX. Cuối cùng, constructor áp dụng chiến lược **"Tải dữ liệu sớm" (Eager Loading)** bằng cách gọi ngay loadBookings() (dòng 39). Điều này đảm bảo trải nghiệm người dùng liền mạch: ngay khi giao diện xuất hiện, dữ liệu đặt bàn đã được tải sẵn sàng mà không cần thêm bất kỳ thao tác kích hoạt nào từ phía người dùng.

3.3.1.3 Method setTableView(): Ràng buộc TableView

```
/**
 * Set the TableView reference.
 */
public void setTableView(TableView<Booking> tableView) {
    this.tableView = tableView;
    tableView.setItems(bookingList);
}
```

Đoạn mã từ dòng 42 đến 46 thực hiện bước "đấu nối" (wiring) quan trọng giữa thành phần hiển thị và dữ liệu nội tại. Phương thức này đóng vai trò là điểm tiếp nhận trong mô hình **Setter Injection**: nó cho phép MainApp (nơi khởi tạo giao diện) chuyển giao quyền kiểm soát đối tượng TableView cho Controller sau khi đối tượng này đã được tạo.

Điểm cốt lõi về mặt kỹ thuật nằm ở dòng 46 với lệnh tableView.setItems(bookingList). Đây không chỉ là một phép gán đơn thuần mà là hành động thiết lập **Ràng buộc dữ liệu (Data Binding)**. Bằng cách liên kết bảng với một ObservableList, ứng dụng tạo ra một luồng dữ liệu tự động: bảng trở thành một "tấm gương" phản chiếu trạng thái của danh sách. Bất kỳ thao tác thêm, xóa, hay sửa đổi nào trên bookingList sẽ ngay lập tức kích hoạt sự kiện vẽ lại giao diện tương ứng, loại bỏ hoàn toàn gánh nặng phải cập nhật UI thủ công.

3.3.1.4 Method setFormFields(): Đăng ký Form Fields và Populate Combo Boxes

```
    /**
     * Set form input fields.
     */
    public void setFormFields(TextField idField, TextField
customerNameField, TextField phoneNumberField,
                              TextField numberOfGuestsField,
DatePicker datePicker,
                              ComboBox<Integer> hourComboBox,
ComboBox<Integer> minuteComboBox,
                              TextField tableIdField,
ComboBox<String> statusComboBox) {
        this.idField = idField;
        this.customerNameField = customerNameField;
        this.phoneNumberField = phoneNumberField;
        this.numberOfGuestsField = numberOfGuestsField;
        this.datePicker = datePicker;
        this.hourComboBox = hourComboBox;
        this.minuteComboBox = minuteComboBox;
        this.tableIdField = tableIdField;
        this.statusComboBox = statusComboBox;

        // Populate hour combo box (0-23)
        for (int i = 0; i < 24; i++) {
            hourComboBox.getItems().add(i);
        }

        // Populate minute combo box (0, 15, 30, 45)
        for (int i = 0; i < 60; i += 15) {
            minuteComboBox.getItems().add(i);
        }

        // Set default values
        hourComboBox.setValue(18);
        minuteComboBox.setValue(0);

        // Populate status combo box
        statusComboBox.getItems().addAll("CONFIRMED",
"SEATED", "CANCELLED");
        statusComboBox.setValue("CONFIRMED");
    }
```

Đoạn mã từ dòng 50 đến 83 thực hiện vai trò **Khởi tạo và Cấu hình Form (Form Initialization & Configuration)**. Thông qua phương thức setFormFields, Controller tiếp nhận quyền kiểm soát hoàn toàn đối với 9 thành phần giao diện từ MainApp. Việc lưu trữ các tham chiếu này (dòng 57-65) là bước đệm thiết yếu, cho phép logic nghiệp vụ sau này có thể "đọc" dữ liệu người dùng nhập hoặc "ghi" dữ liệu ngược lại form khi cần chỉnh sửa.

Điểm sáng của phương thức nằm ở logic nạp dữ liệu (Populate) thông minh cho các ComboBox. Đối với thời gian, thay vì liệt kê từng phút, vòng lặp tạo bước nhảy 15 phút (0, 15, 30, 45) tại dòng 72-75 phản ánh một **quy tắc nghiệp vụ thực tế**: nhà hàng thường quản lý đặt bàn theo khung thời gian cố định (time slots). Điều này vừa giảm tải nhận thức cho người dùng, vừa giúp việc sắp xếp lịch trình phía sau gọn gàng hơn. Bên cạnh đó, việc thiết lập giá trị mặc định (Default Values) là 18:00 (giờ vàng bữa tối) và trạng thái "CONFIRMED" thể hiện tư duy **"Smart Defaults"** trong thiết kế UX, giúp tối ưu hóa tốc độ thao tác cho kịch bản sử dụng phổ biến nhất.

3.3.1.5 Method loadBookings(): Tải dữ liệu từ Service

```
    /**
     * Load all bookings into the table.
     */
    public void loadBookings() {
        bookingList.clear();
        bookingList.addAll(bookingService.getAllBookings());
    }
```

Đoạn mã từ dòng 86 đến 90 định nghĩa phương thức loadBookings, đóng vai trò là cơ chế **đồng bộ hóa dữ liệu (Data Synchronization)** chủ động của Controller. Chiến lược được áp dụng ở đây là "Làm sạch và Nạp lại" (Clean & Reload): dòng 89 dọn sạch danh sách cũ để đảm bảo tính toàn vẹn, hành động này ngay lập tức kích hoạt giao diện bảng xóa rỗng thông qua cơ chế quan sát của ObservableList.

Ngay sau đó, dòng 90 thực hiện thao tác **"Kéo dữ liệu" (Data Pulling)** từ tầng Service thông qua bookingService.getAllBookings(). Việc sử dụng addAll thay vì thêm từng phần tử giúp tối ưu hóa hiệu năng cập nhật giao diện, giảm thiểu số lần render lại của TableView. Về mặt quy trình, phương thức này đóng vai trò then chốt trong vòng đời ứng dụng: nó không chỉ chạy khi khởi động mà còn được tái sử dụng

sau mỗi thao tác CRUD (Thêm/Sửa/Xóa) để đảm bảo người dùng luôn nhìn thấy trạng thái dữ liệu mới nhất ("Single Source of Truth") từ cơ sở dữ liệu.

3.3.1.6 Method handleAdd(): Xử lý thêm Booking

```
    /**
     * Handle add button click.
     */
    public void handleAdd() {
        try {
            Booking booking = createBookingFromForm();
            bookingService.addBooking(booking);
            loadBookings();
            clearForm();
            showSuccessAlert("Booking added successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error adding booking", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 94 đến 105 hiện thực hóa trọn vẹn **luồng xử lý giao dịch thêm mới (Add Transaction Workflow)**. Bắt đầu với lớp bảo vệ try-catch, phương thức thiết lập cơ chế an toàn để đón bắt mọi ngoại lệ từ tầng Service—đặc biệt là IllegalArgumentException khi dữ liệu không thỏa mãn quy tắc nghiệp vụ. Logic xử lý tuân thủ chặt chẽ trình tự chuẩn của một Controller: **Thu thập & Đóng gói** dữ liệu (thông qua helper createBookingFromForm), **Ủy quyền xử lý** cho Service, và cuối cùng là **Đồng bộ hóa Giao diện**.

Điểm nhấn về trải nghiệm người dùng (UX) nằm ở chuỗi hành động hậu kỳ (post-processing): ngay sau khi ghi nhận thành công, hệ thống thực hiện đồng thời việc làm tươi bảng dữ liệu (loadBookings), dọn sạch form nhập liệu (clearForm) và phát tín hiệu xác nhận (showSuccessAlert). Quy trình khép kín này cung cấp phản hồi tức thì, xác nhận hành động đã hoàn tất và đưa giao diện về trạng thái sẵn sàng cho tác vụ tiếp theo mà không cần thao tác thừa.

3.3.1.7 Method handleUpdate(): Xử lý cập nhật Booking

```
    /**
     * Handle update button click.
     */
    public void handleUpdate() {
        try {
            Booking booking = createBookingFromForm();
            if (booking.getId() == null || booking.getId().isEmpty()) {
```

```
                showErrorAlert("Error", "Please select a booking to
update");
                return;
            }
            bookingService.updateBooking(booking);
            loadBookings();
            clearForm();
            showSuccessAlert("Booking updated successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error updating booking", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 109 đến 124 hiện thực hóa quy trình **Cập nhật thông tin (Update Workflow)**, tuy có cấu trúc tương đồng với chức năng Thêm mới, nhưng logic này chứa đựng một bước kiểm tra "tiền điều kiện" (Pre-condition) chí mạng tại dòng 114-117. Đây là một **Guard Clause** (Mệnh đề bảo vệ) điển hình: Controller chủ động chặn đứng thao tác ngay từ giao diện nếu không phát hiện ID hợp lệ (tức là người dùng chưa chọn bản ghi nào từ bảng), ngăn chặn việc gửi yêu cầu cập nhật vô danh xuống tầng Service.

Khi vượt qua rào chắn kiểm tra này, đối tượng Booking được tái tạo từ form và chuyển giao cho bookingService.updateBooking (dòng 118) để xử lý nghiệp vụ tìm kiếm và ghi đè. Chu trình kết thúc bằng bộ ba hành động chuẩn mực tương tự như khi thêm mới: đồng bộ lại dữ liệu (loadBookings), dọn dẹp giao diện (clearForm), và phản hồi trạng thái thành công (showSuccessAlert), đảm bảo tính nhất quán giữa hành động của người dùng và trạng thái lưu trữ của hệ thống.

3.3.1.8 Method handleCancel(): Xử lý hủy Booking

```
    /**
     * Handle cancel button click.
     */
    public void handleCancel() {
        Booking selected = tableView.getSelectionModel().getSelectedItem();
        if (selected == null) {
            showErrorAlert("Error", "Please select a booking to cancel");
            return;
        }

        try {
            bookingService.cancelBooking(selected.getId());
            loadBookings();
            clearForm();
            showSuccessAlert("Booking cancelled successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error cancelling booking", e.getMessage());
```

```
        }
    }
```

Đoạn mã từ dòng 128 đến 144 hiện thực hóa nghiệp vụ **Hủy đặt bàn (Cancellation Workflow)**, một thao tác mang tính định hướng trạng thái (State-oriented). Khác với quy trình nhập liệu của Thêm/Sửa, logic này áp dụng pattern **"Thao tác trên đối tượng được chọn" (Action on Selection)**: Controller tương tác trực tiếp với TableView thông qua getSelectionModel() (dòng 131) để xác định mục tiêu thay vì đọc từ form.

Một lớp bảo vệ (Guard Clause) tại dòng 132-135 đảm bảo tính an toàn: hệ thống từ chối thực thi nếu không có dòng nào được chọn. Điểm tinh gọn về mặt thiết kế nằm ở dòng 138: phương thức chỉ truyền duy nhất **ID** xuống tầng Service (cancelBooking). Điều này phản ánh chính xác bản chất nghiệp vụ: hủy là một thao tác chuyển đổi trạng thái (State Transition) dựa trên định danh, không đòi hỏi payload dữ liệu phức tạp. Quy trình kết thúc bằng việc làm mới danh sách và xóa form, đảm bảo giao diện đồng bộ tức thì với trạng thái nghiệp vụ mới.

3.3.1.9 Method handleSeatCustomer(): Xử lý xác nhận Khách Đã Đến

```
    /**
     * Handle seat customer button click.
     */
    public void handleSeatCustomer() {
        Booking                         selected                         =
tableView.getSelectionModel().getSelectedItem();
        if (selected == null) {
            showErrorAlert("Error", "Please select a booking to seat");
            return;
        }

        try {
            bookingService.seatCustomer(selected.getId());
            loadBookings();
            populateForm(selected);
            showSuccessAlert("Customer seated successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error seating customer", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 148 đến 164 hiện thực hóa quy trình **"Xếp bàn" (Seating Workflow)**, một thao tác chuyển đổi trạng thái quan trọng đánh dấu sự hiện diện thực tế của khách hàng tại nhà hàng. Tương tự như chức năng Hủy, logic này bắt đầu bằng

việc xác thực đối tượng được chọn từ bảng (dòng 151-155), đảm bảo thao tác luôn gắn liền với một đơn đặt hàng cụ thể.

Tuy nhiên, điểm nhấn đặc biệt về UX nằm ở chiến lược phản hồi sau khi xử lý (dòng 160). Thay vì dọn sạch giao diện bằng clearForm() như các thao tác trước, Controller chủ động gọi **populateForm(selected)**. Quyết định thiết kế này mang ý nghĩa nghiệp vụ sâu sắc: khi nhân viên xác nhận "Khách đã ngồi", họ thường cần tiếp tục quan sát chi tiết đặt bàn hoặc thực hiện các tác vụ tiếp theo (như gọi món). Việc giữ lại thông tin trên form và cập nhật trạng thái mới ("SEATED") giúp duy trì ngữ cảnh làm việc (Context Retention), tạo cảm giác liền mạch thay vì ngắt quãng quy trình làm việc của nhân viên.

3.3.1.10 Method handleTableSelection(): Xử lý lựa chọn Hàng trong Bảng

```
    /**
     * Handle table row selection.
     */
    public void handleTableSelection() {
        Booking                    selected                    =
tableView.getSelectionModel().getSelectedItem();
        if (selected != null) {
            populateForm(selected);
        }
    }
```

Đoạn mã từ dòng 168 đến 174 thiết lập cơ chế **Đồng bộ ngược (Reverse Synchronization)** từ bảng hiển thị về form nhập liệu. Phương thức handleTableSelection đóng vai trò là bộ lắng nghe sự kiện điều hướng: ngay khi người dùng tương tác với một dòng dữ liệu, Controller lập tức trích xuất đối tượng Booking tương ứng thông qua SelectionModel.

Logic này hoạt động như một cầu nối thông minh: sau khi xác nhận đối tượng tồn tại (dòng 172), nó không tự xử lý dữ liệu thô mà ủy quyền cho phương thức chuyên biệt populateForm. Điều này hiện thực hóa mô hình tương tác **Master-Detail** (Danh sách Chi tiết) điển hình, cho phép người dùng chuyển đổi mượt mà từ trạng thái "Tra cứu tổng quan" (trên bảng) sang trạng thái "Chi tiết/Chỉnh sửa" (trên form) chỉ bằng một cú nhấp chuột, tạo nên trải nghiệm người dùng liền mạch và trực quan.

3.3.1.11 Method createBookingFromForm(): Tạo Booking Object từ Form

```
    /**
     * Create Booking object from form fields.
     */
    private Booking createBookingFromForm() {
        Booking booking = new Booking();
        booking.setId(idField.getText().trim());
        booking.setCustomerName(customerNameField.getText().trim());
        booking.setPhoneNumber(phoneNumberField.getText().trim());

        try {
            booking.setNumberOfGuests(Integer.parseInt(numberOfGuestsFi
eld.getText().trim()));
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Number of guests must be
a valid integer");
        }

        booking.setDate(datePicker.getValue());

        // Create LocalTime from hour and minute combo boxes
        Integer hour = hourComboBox.getValue();
        Integer minute = minuteComboBox.getValue();
        if (hour != null && minute != null) {
            booking.setStartTime(LocalTime.of(hour, minute));
        }

        booking.setTableId(tableIdField.getText().trim());
        booking.setStatus(statusComboBox.getValue());

        return booking;
    }
```

Đoạn mã từ dòng 178 đến 204 hiện thực hóa quy trình **Chuyển đổi dữ liệu (Data Transformation)** từ các trường giao diện rời rạc sang thực thể Booking chặt chẽ. Phương thức helper này đóng vai trò "vệ sinh" và hợp nhất dữ liệu đầu vào: chuỗi ký tự được làm sạch bằng .trim() để loại bỏ khoảng trắng thừa, trong khi các lỗi định dạng số (như nhập chữ vào trường số lượng khách) được **bọc lại (Exception Wrapping)** thành IllegalArgumentException, giúp bảo vệ tầng Service khỏi dữ liệu rác và cung cấp thông báo lỗi nghiệp vụ rõ ràng.

Điểm nhấn kỹ thuật nằm ở logic **Tổng hợp thời gian (Time Aggregation)**: phương thức gom nhặt dữ liệu từ DatePicker và hai ComboBox (giờ/phút) để tái tạo chính xác các đối tượng LocalDate và LocalTime (dòng 194-199). Đây là chốt chặn

quan trọng đảm bảo tính toàn vẹn và đúng kiểu (type-safe) của dữ liệu trước khi nó rời khỏi tầng Controller.

3.3.1.12 Method populateForm(): Điền Form từ Booking Object

```
    /**
     * Populate form fields with selected booking data.
     */
    private void populateForm(Booking booking) {
        idField.setText(booking.getId());
        customerNameField.setText(booking.getCustomerName());
        phoneNumberField.setText(booking.getPhoneNumber());
        numberOfGuestsField.setText(String.valueOf(booking.getNumberOfG
uests()));
        datePicker.setValue(booking.getDate());

        if (booking.getStartTime() != null) {
            hourComboBox.setValue(booking.getStartTime().getHour());
            minuteComboBox.setValue(booking.getStartTime().getMinute())
;
        }

        tableIdField.setText(booking.getTableId());
        statusComboBox.setValue(booking.getStatus());
    }
```

Đoạn mã từ dòng 208 đến 223 hiện thực hóa quy trình **Chuyển đổi ngược (Reverse Transformation)**: ánh xạ dữ liệu từ Domain Model (Booking) trở lại các thành phần giao diện để phục vụ mục đích xem chi tiết hoặc chỉnh sửa. Các trường văn bản và ngày tháng được gán trực tiếp, trong khi dữ liệu số (như số lượng khách) được chuyển đổi an toàn sang chuỗi thông qua String.valueOf().

Điểm nhấn kỹ thuật nằm ở logic **Phân rã thời gian (Time Decomposition)** tại dòng 217-220: đối tượng LocalTime được tách thành hai phần Giờ và Phút riêng biệt để khớp với cấu trúc hai ComboBox độc lập trên UI. Phương thức này đảm bảo tính đồng bộ hai chiều: dữ liệu không chỉ được đọc từ form mà còn có thể được nạp lại vào form một cách chính xác, tạo nên trải nghiệm người dùng liền mạch khi thao tác với các bản ghi.

3.3.1.13 Method clearForm(): Xóa Form Fields

```
    /**
     * Clear form fields.
     */
```

```
    private void clearForm() {
        idField.clear();
        customerNameField.clear();
        phoneNumberField.clear();
        numberOfGuestsField.clear();
        datePicker.setValue(LocalDate.now());
        hourComboBox.setValue(18);
        minuteComboBox.setValue(0);
        tableIdField.clear();
        statusComboBox.setValue("CONFIRMED");
        if (tableView != null) {
            tableView.getSelectionModel().clearSelection();
        }
    }
```

Đoạn mã từ dòng 227 đến 241 hiện thực hóa quy trình **Tái lập trạng thái (State Reset)** cho giao diện thông qua phương thức clearForm. Không chỉ đơn thuần xóa trắng các trường văn bản, logic này áp dụng chiến lược **"Giá trị mặc định thông minh" (Smart Defaults)** để tối ưu thao tác nhập liệu tiếp theo: ngày được đặt về hiện tại (LocalDate.now()), giờ về khung giờ vàng 18:00, và trạng thái về "CONFIRMED".

Về mặt tương tác, hành động clearSelection() (dòng 240) đóng vai trò ngắt ngữ cảnh: nó loại bỏ trạng thái "đang chọn" trên bảng dữ liệu, báo hiệu trực quan cho người dùng rằng hệ thống đã thoát khỏi chế độ chỉnh sửa và sẵn sàng cho một giao dịch mới hoàn toàn.

3.3.1.14 Method showSuccessAlert(): Hiển thị Thông báo Thành công

```
    /**
     * Show success alert.
     */
    private void showSuccessAlert(String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Success");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 245 đến 252 xây dựng phương thức `showSuccessAlert`, một thành phần UX thiết yếu giúp xác nhận kết quả thao tác. Sử dụng `Alert.AlertType.INFORMATION`, phương thức tạo ra một hộp thoại chuẩn với biểu tượng thông tin, truyền tải trạng thái tích cực một cách trực quan. Điểm mấu chốt về mặt kỹ thuật là lệnh `showAndWait()` (dòng 252): đây là cơ chế hiển thị **chặn (blocking)**, buộc luồng ứng dụng phải tạm dừng và chờ người dùng xác nhận (đóng hộp thoại) trước khi tiếp tục. Điều này đảm bảo thông điệp thành công luôn được người dùng ghi nhận, tạo cảm giác an tâm và chắc chắn trong quy trình sử dụng.

3.3.1.15 Method showErrorAlert(): Hiển thị Thông báo Lỗi

```
    /**
     * Show error alert.
     */
    private void showErrorAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```


Đoạn mã từ dòng 256 đến 263 hoàn thiện cơ chế phản hồi của ứng dụng với phương thức showErrorAlert. Tương tự như thông báo thành công, nhưng phương thức này sử dụng Alert.AlertType.ERROR (dòng 259) để kích hoạt giao diện cảnh báo nghiêm trọng (thường đi kèm biểu tượng dấu X đỏ), báo hiệu trực quan cho người dùng rằng quy trình đã bị gián đoạn.

Điểm khác biệt trong thiết kế nằm ở tham số title (dòng 260), cho phép tùy biến tiêu đề hộp thoại để phù hợp với ngữ cảnh lỗi cụ thể (ví dụ: "Lỗi Nhập liệu" hay "Lỗi Kết nối"). Cơ chế showAndWait() tiếp tục được áp dụng để đảm bảo thông báo lỗi là một **chặn (blocking)** bắt buộc, buộc người dùng phải nhận thức được vấn đề trước khi quay lại giao diện chính để thực hiện các sửa đổi cần thiết.

3.3.1.16 Method getBookingList(): Getter cho Observable List

```
    /**
     * Get observable list for binding.
     */
```

```
    public ObservableList<Booking> getBookingList() {
        return bookingList;
    }
```

Đoạn mã từ dòng 267 đến 270 định nghĩa phương thức `getBookingList`, đóng vai trò là "cổng kết nối" dữ liệu ra bên ngoài. Bằng cách trả về **tham chiếu trực tiếp (Reference)** của đối tượng `bookingList` thay vì một bản sao, Controller cho phép `MainApp` thiết lập cơ chế **Ràng buộc dữ liệu (Data Binding)** với `TableView`. Chiến lược "Công khai trạng thái nội bộ" (Exposing Internal State) này là chìa khóa để mô hình Observable hoạt động hiệu quả: Controller giữ quyền kiểm soát logic nghiệp vụ (Thêm/Sóa/Sửa), trong khi Giao diện (View) tự động phản ánh các thay đổi đó theo thời gian thực mà không cần sự can thiệp thủ công từ lớp cha.

3.3.1.17 Tổng kết BookingController

BookingController thể hiện các **Design Patterns** và nguyên tắc thiết kế quan trọng:

- **Separation of Concerns**: Controller tách biệt **UI Logic** (đọc/ghi form fields) khỏi **Business Logic** (được xử lý trong Service).
- **Dependency on Service**: Controller phụ thuộc vào BookingService để thực thi logic, không truy cập trực tiếp vào **Repository**.
- **Observable Pattern**: Sử dụng ObservableList để tự động đồng bộ dữ liệu giữa Controller và UI (**Data Binding**).
- **Error Handling**: Sử dụng khối try-catch để xử lý **Exception** từ Service và hiển thị thông báo lỗi (Alert) cho người dùng.
- **Data Transformation**: Chuyển đổi dữ liệu giữa **UI Format** (TextField, ComboBox) và **Domain Format** (Booking object).
- **Form Management**: Quản lý trạng thái form (populate, clear) để tối ưu hóa **UX**.
- **Business Operations**: Xử lý cả **CRUD Operations** tiêu chuẩn và các nghiệp vụ đặc thù (cancel, seat customer).

Tổng kết: Controller này đóng vai trò là cầu nối giữa **UI Layer** (MainApp) và **Service Layer** (BookingService), đảm bảo kiến trúc **Loose Coupling** và dễ bảo trì.

3.3.2 Phân tích EmployeeController.java

'EmployeeController' là controller chịu trách nhiệm quản lý các tương tác liên quan đến nhân viên trong hệ thống nhà hàng. Controller này xử lý các thao tác CRUD cơ bản (Create, Read, Update, Delete) cho entity Employee. Điểm đặc biệt của controller này là hỗ trợ cả hai pattern khởi tạo: tự tạo service instance hoặc nhận service qua dependency injection, cho phép chia sẻ service instance giữa nhiều controller.

3.3.2.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
1  package com.restaurantmanagement.controller;
2
3  import com.restaurantmanagement.model.Employee;
4  import com.restaurantmanagement.service.EmployeeService;
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.scene.control.Alert;
8  import javafx.scene.control.TableView;
9  import javafx.scene.control.TextField;
10
11 import java.util.List;
```

Đoạn mã từ dòng 1 đến 11 thiết lập ngữ cảnh hoạt động cho EmployeeController. Sau khai báo package chuẩn (dòng 1), các dòng 3-4 import Employee và EmployeeService, xác lập mối quan hệ **Dependency** trực tiếp giữa Controller với **Model** và **Service Layer**.

Các dòng 5-9 cung cấp các thành phần cốt lõi của JavaFX: ObservableList để hỗ trợ **Data Binding**, cùng các UI controls (Alert, TableView, TextField) phục vụ hiển thị và tương tác. Dòng 11 import java.util.List, tuy chưa được sử dụng (unused import), nhưng có thể nhằm dự phòng cho các thao tác collection trong tương lai. Tổng thể, phần header này gọn nhẹ hơn nhiều so với BookingController do nghiệp vụ quản lý nhân viên không yêu cầu xử lý phức tạp về Date/Time hay các logic **Combo Box** lồng nhau.

b) Khai báo Class và JavaDoc

```
/**
 * Controller for Employee management UI.
 * Handles user interactions and updates the view.
 */
public class EmployeeController {
```

Đoạn mã từ dòng 13 đến 17 định nghĩa vai trò và phạm vi hoạt động của EmployeeController. Phần **JavaDoc** xác nhận trách nhiệm cốt lõi của class: quản lý **UI** cho phân hệ nhân viên, đóng vai trò trung gian xử lý các **User Interactions** và cập nhật **View** tương ứng.

Khai báo public class là bắt buộc để đảm bảo tính khả dụng (accessibility), cho phép MainApp hoặc các container khác khởi tạo và truy cập Controller này. Cấu trúc này tuân thủ chuẩn thiết kế đồng nhất với BookingController, đảm bảo tính nhất quán trong toàn bộ kiến trúc ứng dụng.

c) Khai báo Field Service và Data

```
private final EmployeeService employeeService;
private final ObservableList<Employee> employeeList;
private TableView<Employee> tableView;
```

Đoạn mã từ dòng 18 đến 20 thiết lập các thành phần cốt lõi quản lý trạng thái và phụ thuộc của Controller.

- **Dòng 18**: Khai báo employeeService với từ khóa final, khẳng định tính **Immutability** của reference tới Service Layer. Điều này đảm bảo logic nghiệp vụ được xử lý bởi một instance duy nhất và không thay đổi trong suốt vòng đời của Controller.
- **Dòng 19**: employeeList đóng vai trò là Model trong pattern MVC của JavaFX. Việc sử dụng ObservableList thiết lập cơ chế **Data Binding** tự động, giúp đồng bộ hóa dữ liệu giữa bộ nhớ và giao diện người dùng.
- **Dòng 20**: tableView lưu trữ tham chiếu đến thành phần UI hiển thị danh sách, được định kiểu Generic <Employee> để đảm bảo **Type Safety** khi thao tác với các dòng dữ liệu.

d) Khai báo Field UI Components

```
// UI Components (for form inputs)
```

```
    private TextField idField;
    private TextField nameField;
    private TextField positionField;
    private TextField phoneField;
    private TextField emailField;
```

Đoạn mã từ dòng 22 đến 27 khai báo các **UI Components** cho form nhập liệu, bao gồm 5 `TextField` map trực tiếp với các thuộc tính của `Employee` Model. Trong đó, `idField` đóng vai trò hiển thị định danh (thường là read-only), còn `nameField`, `positionField`, `phoneField` và `emailField` phục vụ việc nhập liệu thông tin chi tiết.

So với `BookingController`, cấu trúc form này đơn giản và "phẳng" hơn nhiều. Nó thuần túy dựa trên **Text Input**, loại bỏ hoàn toàn sự phức tạp của các controls chuyên dụng như `DatePicker` hay `ComboBox`, phản ánh đúng bản chất dữ liệu tĩnh của đối tượng nhân viên.

3.3.2.2 Constructor và khởi tạo

a) Constructor không tham số

```
    public EmployeeController() {
        this.employeeService = new EmployeeService();
        this.employeeList = FXCollections.observableArrayList();
        loadEmployees();
    }
```

Đoạn mã từ dòng 29 đến 32 định nghĩa **Constructor** của class, nơi thiết lập trạng thái khởi đầu cho Controller. Tại dòng 30, EmployeeService được khởi tạo trực tiếp thông qua từ khóa new. Cách tiếp cận này tạo ra một **Direct Dependency** (Phụ thuộc trực tiếp), trong đó Controller tự quản lý vòng đời của Service instance thay vì nhận từ bên ngoài (Dependency Injection), phù hợp với mô hình ứng dụng đơn giản.

Ngay sau đó, employeeList được khởi tạo dưới dạng ObservableList rỗng (dòng 31) để thiết lập cơ sở cho **Data Binding**. Cuối cùng, phương thức loadEmployees() được gọi ngay lập tức (Eager Loading) tại dòng 32, đảm bảo dữ liệu được tải sẵn sàng vào bộ nhớ ngay khi Controller hoàn tất việc khởi tạo.

b) Constructor với Dependency Injection

```
    /**
     * Constructor with shared EmployeeService instance.
     */
    public
EmployeeController(com.restaurantmanagement.service.EmployeeS
ervice employeeService) {
        this.employeeService = employeeService;
        this.employeeList                                    =
FXCollections.observableArrayList();
        loadEmployees();
    }
```

- Đoạn mã từ dòng 35 đến 41 triển khai kỹ thuật **Constructor Injection** thông qua cơ chế **Overloading**. Khác với constructor mặc định, phương thức này chấp nhận một instance `EmployeeService` từ bên ngoài, thay vì tự khởi tạo mới.
- Việc sử dụng **Fully Qualified Name** (`com.restaurantmanagement.service.EmployeeService`) tại dòng 38 tuy dài dòng nhưng đảm bảo độ chính xác tuyệt đối về namespace, tránh mọi nhầm lẫn tiềm ẩn.
- Giá trị cốt lõi của thiết kế này nằm ở tính linh hoạt:
    - **Shared State Management**: Cho phép chia sẻ một instance Service duy nhất giữa các Controller (ví dụ: giữa quản lý Nhân viên và quản lý Ca làm việc), đảm bảo tính nhất quán dữ liệu toàn cục.
    - **Testability**: Tạo điều kiện thuận lợi cho Unit Testing bằng cách cho phép inject các **Mock Object** thay vì Service thực tế.

3.3.2.3 Method setTableView(): Ràng buộc TableView

```
    /**
     * Set the TableView reference.
     */
    public void setTableView(TableView<Employee> tableView) {
        this.tableView = tableView;
        tableView.setItems(employeeList);
    }
```

Đoạn mã từ dòng 44 đến 48 thực hiện việc cấu hình thành phần hiển thị chính thông qua **Reference Injection**. Phương thức setTableView tiếp nhận tham chiếu của đối tượng TableView từ lớp khởi tạo (MainApp) và lưu trữ vào biến cục bộ.

Trọng tâm kỹ thuật nằm ở dòng 48: tableView.setItems(employeeList). Lệnh này thiết lập cơ chế **Data Binding** giữa Model (ObservableList) và View (TableView). Nhờ đó, Controller không cần viết logic cập nhật giao diện thủ công; mọi thao tác thêm/xóa/sửa trên danh sách employeeList sẽ tự động kích hoạt sự kiện vẽ lại (repaint) trên bảng. Cấu trúc này sao chép chính xác pattern đã dùng trong BookingController, đảm bảo tính **Nhất quán (Consistency)** cho toàn bộ tầng Presentation.

3.3.2.4 Method setFormFields(): Đăng ký Form Fields

```
    /**
     * Set form input fields.
     */
    public void setFormFields(TextField idField, TextField
nameField, TextField positionField,
                              TextField phoneField, TextField
emailField) {
        this.idField = idField;
        this.nameField = nameField;
        this.positionField = positionField;
        this.phoneField = phoneField;
        this.emailField = emailField;
    }
```

Đoạn mã từ dòng 52 đến 59 hiện thực hóa việc **Inject** các thành phần giao diện vào Controller. Phương thức setFormFields tiếp nhận 5 tham chiếu TextField từ MainApp và gán trực tiếp vào các biến instance cục bộ, cấp quyền cho Controller đọc và ghi dữ liệu trên Form.

So với BookingController, phương thức này thể hiện sự **Tinh giản (Simplification)** rõ rệt. Do đặc thù dữ liệu nhân viên chỉ gồm văn bản thuần túy (không có danh mục chọn hay ngày tháng phức tạp), logic tại đây là **Setter thuần túy**, loại bỏ hoàn toàn các bước khởi tạo dữ liệu ban đầu (như setItems cho ComboBox hay setValue cho DatePicker).

3.3.2.5 Method loadEmployees(): Tải Dữ liệu từ Service

```
**
 * Load all employees into the table.
 */
public void loadEmployees() {
    employeeList.clear();
    employeeList.addAll(employeeService.getAllEmployees());
}
```

Đoạn mã từ dòng 64 đến 68 thực hiện quy trình **Data Loading** chuẩn mực. Phương thức bắt đầu bằng employeeList.clear() để làm sạch trạng thái hiện tại, ngăn chặn việc hiển thị dữ liệu cũ hoặc trùng lặp. Ngay sau đó, employeeList.addAll() thực hiện cập nhật hàng loạt (batch update) dữ liệu mới nhất được truy xuất từ EmployeeService.

Quy trình này đồng bộ hoàn toàn với pattern đã thấy trong BookingController, đảm bảo tính **Nhất quán (Consistency)** trong cách ứng dụng phản ánh dữ liệu từ Backend lên UI.

3.3.2.6 Method handleAdd(): Xử lý Thêm Employee

```
/**
 * Handle add button click.
 */
public void handleAdd() {
    try {
        Employee employee = createEmployeeFromForm();
        employeeService.addEmployee(employee);
        loadEmployees();
        clearForm();
        showSuccessAlert("Employee added successfully!");
    } catch (IllegalArgumentException e) {
        showErrorAlert("Error adding employee", e.getMessage());
    }
}
```

Đoạn mã từ dòng 72 đến 82 tái hiện chính xác cấu trúc **Event Handler** tiêu chuẩn đã được thiết lập ở `BookingController`. Quy trình xử lý tuân thủ nghiêm ngặt nguyên tắc an toàn: toàn bộ logic được bao bọc trong khối `try-catch` để kiểm soát ngoại lệ. Dữ liệu từ form trước tiên được đóng gói thành đối tượng thông qua `createEmployeeFromForm()` (dòng 76), sau đó được chuyển xuống tầng Service để xử lý lưu trữ (dòng 77).

Chuỗi hành động kết thúc bằng việc cập nhật toàn diện giao diện: `loadEmployees()` làm mới danh sách, `clearForm()` dọn dẹp ô nhập liệu, và `showSuccessAlert()` cung cấp phản hồi xác nhận. Sự tương đồng tuyệt đối về luồng xử lý này minh chứng cho tính **Chuẩn hóa (Standardization)** cao trong kiến trúc phần mềm, giúp việc bảo trì và mở rộng trở nên dễ dàng.

3.3.2.7 Method handleUpdate(): Xử lý Cập nhật Employee

```
        /**
         * Handle update button click.
         */
        public void handleUpdate() {
            try {
                Employee employee = createEmployeeFromForm();
                if (employee.getId() == null || employee.getId().isEmpty())
{
                    showErrorAlert("Error", "Please select an employee to
update");
                    return;
                }
                employeeService.updateEmployee(employee);
                loadEmployees();
                clearForm();
                showSuccessAlert("Employee updated successfully!");
            } catch (IllegalArgumentException e) {
                showErrorAlert("Error updating employee", e.getMessage());
            }
        }
```

Đoạn mã từ dòng 87 đến 101 thực hiện nghiệp vụ **Update**, tuân thủ quy trình bảo vệ dữ liệu tương tự như `BookingController`. Điểm kiểm soát quan trọng nhất nằm ở logic **Validation ID** (dòng 92-95): hệ thống chặn ngay lập tức nếu trường ID rỗng, đảm bảo người dùng đang thực sự thao tác trên một bản ghi hiện hữu đã được chọn từ bảng, tránh tình trạng "cập nhật ma" (update dữ liệu chưa tồn tại).

Sau khi vượt qua bước kiểm tra, phương thức ủy quyền cho `EmployeeService` thực thi thay đổi xuống cơ sở dữ liệu. Ngay lập tức, chuỗi hành động đồng bộ hóa giao diện được kích hoạt: tải lại danh sách (`loadEmployees`), xóa sạch form (`clearForm`) và thông báo thành công, đảm bảo tính nhất quán giữa dữ liệu backend và hiển thị frontend.

3.3.2.8 Method handleDelete(): Xử lý Xóa Employee

```
    /**
     * Handle delete button click.
     */
    public void handleDelete() {
        Employee selected = tableView.getSelectionModel().getSelectedItem();
        if (selected == null) {
            showErrorAlert("Error", "Please select an employee to delete");
            return;
        }

        try {
            employeeService.deleteEmployee(selected.getId());
            loadEmployees();
            clearForm();
            showSuccessAlert("Employee deleted successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error deleting employee", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 106 đến 122 xử lý nghiệp vụ **Hard Delete** (Xóa vĩnh viễn). Logic bắt đầu bằng việc xác định đối tượng mục tiêu thông qua `SelectionModel` (dòng 109). Validation tại dòng 110-113 đóng vai trò **Guard Clause**, chặn ngay lập tức hành động nếu người dùng chưa chọn dòng nào trên `TableView`.

Khác với `BookingController` sử dụng chiến lược "Soft Delete" (Hủy/Đổi trạng thái) để lưu vết lịch sử, phương thức này gọi `employeeService.deleteEmployee()` để loại bỏ hoàn toàn bản ghi khỏi hệ thống. Điều này phản ánh sự khác biệt trong **Entity Lifecycle**: đơn đặt bàn cần lưu trữ lịch sử hủy, trong khi nhân viên khi bị xóa thường đồng nghĩa với việc loại bỏ khỏi danh sách quản lý hiện hành. Sau khi xóa thành công, UI được làm mới hoàn toàn để đồng bộ với Database.

3.3.2.9 Method handleTableSelection(): Xử lý Lựa chọn Hàng trong Bảng

```
    /**
     * Handle table row selection.
     */
    public void handleTableSelection() {
        Employee                    selected                    =
tableView.getSelectionModel().getSelectedItem();
        if (selected != null) {
            populateForm(selected);
        }
    }
```

Đoạn mã từ dòng 126 đến 132 thực hiện logic **Master-Detail Interaction**. Phương thức handleTableClick đóng vai trò lắng nghe sự kiện từ TableView (Master) và cập nhật chi tiết vào Form nhập liệu (Detail).

Logic cốt lõi dựa trên getSelectionModel().getSelectedItem() để truy xuất đối tượng ngữ cảnh. Điều kiện kiểm tra if (selected != null) là bắt buộc để ngăn lỗi NullPointerException khi người dùng click vào vùng trống của bảng. Khi một dòng hợp lệ được chọn, populateForm() được kích hoạt để ánh xạ dữ liệu ngược trở lại các trường nhập liệu, giúp người dùng dễ dàng xem hoặc sửa đổi thông tin mà không cần nhập lại từ đầu. Đây là pattern UI tiêu chuẩn, đồng bộ hoàn toàn với BookingController.

3.3.2.10 Method createEmployeeFromForm(): Tạo Employee Object từ Form

```
    /**
     * Create Employee object from form fields.
     */
    private Employee createEmployeeFromForm() {
        Employee employee = new Employee();
        employee.setId(idField.getText().trim());
        employee.setName(nameField.getText().trim());
        employee.setPosition(positionField.getText().trim());
        employee.setPhoneNumber(phoneField.getText().trim());
        employee.setEmail(emailField.getText().trim());
        return employee;
    }
```

Đoạn mã từ dòng 136 đến 146 đóng vai trò là một **Factory Method** nội bộ, chịu trách nhiệm đóng gói dữ liệu thô từ giao diện (`View`) thành đối tượng `Employee` hoàn chỉnh (`Model`).

So với `BookingController`, logic tại đây được tối giản hóa tối đa. Do đặc thù dữ liệu nhân viên chủ yếu là văn bản (String), phương thức loại bỏ hoàn toàn gánh nặng của việc **Type Parsing** (chuyển đổi kiểu số hay thời gian) và xử lý ngoại lệ định dạng. Việc áp dụng `.trim()` nhất quán cho mọi trường đầu vào là một thực hành tốt (**Best Practice**) để loại bỏ khoảng trắng thừa, đảm bảo tính sạch sẽ của dữ liệu trước khi đẩy xuống tầng Service.

3.3.2.11 Method populateForm(): Điền Form từ Employee Object

```
    /**
     * Populate form fields with selected employee data.
     */
    private void populateForm(Employee employee) {
        idField.setText(employee.getId());
        nameField.setText(employee.getName());
        positionField.setText(employee.getPosition());
        phoneField.setText(employee.getPhoneNumber());
        emailField.setText(employee.getEmail());
    }
```

Đoạn mã từ dòng 149 đến 156 thực hiện chiều ngược lại của quy trình binding: **Data Population** (Đổ dữ liệu). Phương thức này ánh xạ trực tiếp các thuộc tính của đối tượng `Employee` lên các `TextField` thông qua phương thức `setText()`.

Đây là ví dụ điển hình của **Flat Mapping** (Ánh xạ phẳng). Khác với `BookingController` phải xử lý logic phức tạp để phân tách `LocalTime` vào ComboBox hay chuyển đổi `LocalDate`, ở đây logic hoàn toàn tuyến tính 1:1. Sự đơn giản này xuất phát từ bản chất dữ liệu của `Employee` chủ yếu là chuỗi ký tự, giúp giảm thiểu chi phí xử lý và rủi ro lỗi định dạng trên UI.

3.3.2.12 Method clearForm(): Xóa Form Fields

```
    /**
     * Clear form fields.
     */
    private void clearForm() {
        idField.clear();
        nameField.clear();
        positionField.clear();
```

```
        phoneField.clear();
        emailField.clear();
        if (tableView != null) {
            tableView.getSelectionModel().clearSelection();
        }
    }
```

Đoạn mã từ dòng 160 đến 170 thực hiện chức năng **State Reset** (Đặt lại trạng thái) cho giao diện. Logic tập trung vào việc xóa trắng toàn bộ các `TextField` thông qua phương thức `.clear()` và giải phóng dòng đang được chọn trên bảng (`clearSelection`), đưa form về trạng thái sẵn sàng cho thao tác thêm mới. So với `BookingController`, phương thức này tinh gọn hơn đáng kể do không phải xử lý các giá trị mặc định (default values) cho những control phức tạp như DatePicker hay ComboBox, phản ánh đúng cấu trúc dữ liệu đơn giản của đối tượng nhân viên.

3.3.2.13 Method showSuccessAlert(): Hiển thị Thông báo Thành công

```
    /**
     * Show success alert.
     */
    private void showSuccessAlert(String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Success");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 174 đến 182 thực hiện chức năng hiển thị thông báo thành công với cấu trúc JavaDoc và chữ ký hàm hoàn toàn tương đồng với `BookingController.showSuccessAlert()`. Logic bên trong khởi tạo một `Alert` kiểu `INFORMATION`, thiết lập tiêu đề, nội dung và hiển thị dưới dạng modal (`showAndWait`). Do sự trùng lặp hoàn toàn về mã nguồn này, đây là ứng viên sáng giá để tách thành một **Utility Class** dùng chung, giúp tuân thủ nguyên tắc DRY (Don't Repeat Yourself) và loại bỏ sự dư thừa code giữa các Controller.

3.3.2.14 Method showSuccessAlert(): Hiển thị Thông báo Thành công

```
    /**
     * Show error alert.
     */
    private void showErrorAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 185 đến 193 thực hiện chức năng hiển thị cảnh báo lỗi, với JavaDoc và chữ ký hàm hoàn toàn tương đồng với BookingController.showErrorAlert(). Logic triển khai bên trong khởi tạo một Alert kiểu ERROR, thiết lập tiêu đề, nội dung và hiển thị hộp thoại chờ xác nhận. Sự trùng lặp mã nguồn tuyệt đối này củng cố nhận định rằng logic hiển thị thông báo nên được tách ra thành một **Utility Method** dùng chung, giúp tối ưu hóa cấu trúc và giảm thiểu sự dư thừa trong toàn bộ dự án.

3.3.2.15 Method getEmployeeList(): Getter cho Observable List

```
    /**
     * Get observable list for binding.
     */
    public ObservableList<Employee> getEmployeeList() {
        return employeeList;
    }
```

Đoạn mã từ dòng 196 đến 199 thực hiện chức năng **Accessor** (Getter) tiêu chuẩn, trả về tham chiếu trực tiếp đến đối tượng employeeList. Mục đích chính của phương thức này là phục vụ cơ chế **External Binding**: nó cho phép lớp quản lý chính (MainApp) truy cập vào nguồn dữ liệu ObservableList của Controller để thiết lập kết nối với TableView. Thiết kế này sao chép chính xác API của BookingController, đảm bảo tính đồng bộ trong cách các thành phần ứng dụng giao tiếp với nhau.

3.3.2.16 Tổng kết

EmployeeController hiện thực hóa các mẫu thiết kế (design patterns) và nguyên tắc kiến trúc tương đồng với BookingController, đồng thời thể hiện những đặc thù riêng biệt phù hợp với ngữ cảnh nghiệp vụ thông qua các khía cạnh sau:

- **Cơ chế Khởi tạo Linh hoạt (Constructor Overloading & Dependency Injection)** Lớp áp dụng kỹ thuật nạp chồng phương thức khởi tạo (Constructor Overloading) nhằm đa dạng hóa chiến lược khởi tạo (Initialization Strategy):
  - **Self-contained Instantiation:** Constructor không tham số cho phép lớp tự khởi tạo instance của Service khi hoạt động độc lập.
  - **External Dependency Injection:** Constructor có tham số hỗ trợ tiêm phụ thuộc, cho phép tiếp nhận EmployeeService từ bên ngoài, tạo tiền đề cho việc quản lý vòng đời đối tượng tập trung.
- **Tối giản hóa Logic Quản lý Giao diện (Simplified UI Logic)** Logic quản lý Form được tinh giản đáng kể so với BookingController. Do đặc thù dữ liệu thực thể Employee chủ yếu định dạng văn bản (Text-based), Controller loại bỏ được sự phức tạp của việc xử lý các thành phần giao diện nâng cao (như DatePicker, ComboBox), từ đó giảm tải chi phí tính toán trong quá trình chuyển đổi dữ liệu (Data Transformation).
- **Tính Nhất quán Kiến trúc (Architectural Consistency)** Controller tuân thủ nghiêm ngặt các chuẩn mực thiết kế hệ thống (System Convention):
  - **Data Binding:** Đồng bộ hóa dữ liệu thời gian thực giữa Model và View thông qua ObservableList.
  - **Exception Handling:** Kiểm soát luồng lỗi chặt chẽ bằng khối try-catch nhằm đảm bảo tính ổn định của ứng dụng.
  - **Data Mapping:** Chuẩn hóa quy trình ánh xạ dữ liệu hai chiều giữa tầng giao diện (UI) và đối tượng nghiệp vụ (Domain Object).
  - **State Management:** Quản lý vòng đời trạng thái của Form (populate, reset/clear) theo quy trình chuẩn.
- **Nhận diện Dư thừa và Tiềm năng Tối ưu (Redundancy & Refactoring)** Sự trùng lặp mã nguồn (Code Duplication) được phát hiện tại các phương thức

hiển thị thông báo (showSuccessAlert, showErrorAlert). Theo nguyên tắc DRY (Don't Repeat Yourself), các phương thức này cần được tách biệt và đóng gói vào một lớp tiện ích chung (Utility Class/Helper Class) để tối ưu hóa khả năng bảo trì.

– **Quản lý Trạng thái Chia sẻ (Shared State Management)** Cơ chế Dependency Injection cho phép MainApp điều phối và chia sẻ một instance EmployeeService duy nhất (Singleton-like scope) giữa EmployeeController và ShiftController. Kiến trúc này đảm bảo tính toàn vẹn dữ liệu (Data Integrity) và đồng bộ hóa trạng thái trên toàn bộ ứng dụng.

Kết luận: EmployeeController đóng vai trò là lớp trung gian (Mediator) hiệu quả kết nối Tầng Giao diện (Presentation Layer) và Tầng Dịch vụ (Service Layer). Thiết kế của lớp đạt được sự cân bằng tối ưu giữa tính đơn giản—phù hợp với cấu trúc dữ liệu của thực thể—và tính chặt chẽ trong kiến trúc tổng thể của hệ thống.

3.3.3 Phân tích InventoryController.java

'InventoryController' là controller phức tạp nhất trong hệ thống, chịu trách nhiệm quản lý cả inventory items (hàng tồn kho) và inventory transactions (giao dịch nhập/xuất kho). Controller này xử lý các thao tác CRUD cho items, quản lý stock operations (nhập/xuất), và cung cấp chức năng lọc/tìm kiếm items. Điểm đặc biệt của controller này là quản lý hai entities khác nhau (InventoryItem và InventoryTransaction) với hai TableView riêng biệt.

3.3.3.1 Cấu trúc Class và khai báo Field

a) Package và Import Statement

```
package com.restaurantmanagement.controller;

import com.restaurantmanagement.model.InventoryItem;
import com.restaurantmanagement.model.InventoryTransaction;
import com.restaurantmanagement.service.InventoryService;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.collections.transformation.FilteredList;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
import javafx.scene.control.TableView;
```

```
import javafx.scene.control.TextField;

import java.util.List;
```

Đoạn mã từ dòng 1 đến dòng 14 thiết lập ngữ cảnh hoạt động của InventoryController, hé lộ một sự nâng cấp đáng kể về độ phức tạp so với các bộ điều khiển (Controller) trước đó thông qua hai đặc điểm kỹ thuật chính:

- **Quản lý Đa Thực thể (Multi-Entity Domain Management)** Khác với mô hình "đơn trách nhiệm" (Single Responsibility một controller quản lý một entity) thấy ở EmployeeController, việc nhập khẩu đồng thời InventoryItem và InventoryTransaction (dòng 3-4) cho thấy controller này đảm nhiệm vai trò **Composite Management**. Nó phải xử lý mối quan hệ tương hỗ giữa trạng thái tĩnh (Tồn kho Stock) và dòng chảy dữ liệu (Giao dịch Flow/Transaction), đòi hỏi logic nghiệp vụ phức tạp hơn để đảm bảo tính nhất quán dữ liệu.
- **Chiến lược Thao tác Dữ liệu Động (Dynamic Data Manipulation)** Sự xuất hiện của lớp FilteredList (dòng 8) bên cạnh ObservableList đánh dấu sự chuyển dịch từ hiển thị dữ liệu tĩnh sang tương tác dữ liệu động.
    - **Về mặt kỹ thuật:** FilteredList hoạt động như một lớp vỏ bọc (wrapper) bao quanh danh sách gốc, cho phép áp dụng các vị từ (predicates) để lọc tập hợp con dữ liệu hiển thị mà không làm biến đổi cấu trúc dữ liệu nguồn.
    - **Về mặt chức năng:** Điều này ám chỉ rằng giao diện Kho hàng sẽ tích hợp các tính năng **Tìm kiếm thời gian thực (Real-time Search)** hoặc bộ lọc nghiệp vụ phức tạp, một yêu cầu thiết yếu khi quản lý danh mục hàng hóa số lượng lớn.

b) Khai báo class và Javadoc

```
/**
 * Controller for Inventory management UI.
 * Handles user interactions and updates the view.
 */
public class InventoryController {
```

Đoạn mã từ dòng 16 đến 20 thiết lập nền tảng cấu trúc cho module Kho hàng:

- **Tài liệu kỹ thuật (Documentation):** Khối JavaDoc tuân thủ chuẩn mực dự án, định nghĩa rõ ràng trách nhiệm kép của Controller: quản lý trạng thái tĩnh (Items) và dòng chảy vật tư (Transactions).
- **Khả năng tiếp cận (Accessibility):** Khai báo `public` là điều kiện tiên quyết để cơ chế **Reflection** của `FXMLLoader` hoạt động, cho phép framework tự động khởi tạo và liên kết Controller với file FXML tương ứng.

c) Khai báo Field Service và data

```
    private final InventoryService inventoryService;
    private     final     ObservableList<InventoryItem>
itemList;
    private  final  ObservableList<InventoryTransaction>
transactionList;
    private TableView<InventoryItem> itemTableView;
    private             TableView<InventoryTransaction>
transactionTableView;
```

Đoạn mã từ dòng 21 đến 25 xác lập kiến trúc **Multi-View Controller**, nơi một bộ điều khiển chịu trách nhiệm điều phối hai luồng dữ liệu riêng biệt nhưng có quan hệ mật thiết về mặt nghiệp vụ:

- **Phân tách Mô hình Dữ liệu (Data Model Segregation)** Khác với các Controller trước đó chỉ duy trì một `ObservableList`, lớp này quản lý hai danh sách song song:
  - **`itemList` (State Data):** Lưu trữ trạng thái hiện tại (Snapshot) của kho hàng (số lượng, giá cả).
  - **`transactionList` (Event Data):** Lưu trữ dòng sự kiện (Log) nhập/xuất. Việc tách biệt này tuân thủ nguyên tắc **Separation of Concerns (SoC)**, ngăn chặn việc trộn lẫn logic quản lý tồn kho với logic truy vết lịch sử.
- **Kiến trúc Giao diện Đa Bảng (Multi-Table Architecture)** Sự hiện diện của hai đối tượng `TableView` (`itemTableView` và `transactionTableView`) khẳng định giao diện người dùng sẽ được chia thành hai khu vực chức năng rõ rệt. Controller đóng vai trò là "nhạc trưởng",

đảm bảo khi một hành động xảy ra (ví dụ: Nhập kho), dữ liệu phải được cập nhật đồng bộ trên cả hai bảng: tăng số lượng trong `itemTableView` và thêm dòng nhật ký vào `transactionTableView`.

d) Khai báo Field UI Components Item Form

```
// UI Components (for form inputs)
private TextField idField;
private TextField nameField;
private ComboBox<String> categoryComboBox;
private ComboBox<String> unitComboBox;
private TextField quantityField;
private TextField minimumThresholdField;
private TextField supplierNameField;
private ComboBox<String> storageLocationComboBox;
```

- **Chuẩn hóa Dữ liệu Đầu vào (Input Normalization)** Sự xuất hiện dày đặc của các `ComboBox` (`category`, `unit`, `storageLocation`) thay vì `TextField` tự do là một quyết định thiết kế quan trọng.
    - **Mục đích:** Ép buộc người dùng chọn từ danh sách định sẵn (Predefined List).
    - **Lợi ích:** Đảm bảo tính nhất quán của dữ liệu (Data Consistency), ngăn chặn các biến thể nhập liệu sai lệch (ví dụ: "kg", "kgs", "Kilogram") gây khó khăn cho việc thống kê và báo cáo sau này.
- **Độ phức tạp về Xử lý Dữ liệu (Data Processing Complexity)** Đúng như bạn nhận định, form này phức tạp hơn `EmployeeController` đáng kể:
    - **Parsing:** Controller sẽ phải chịu trách nhiệm chuyển đổi chuỗi (`String`) từ `quantityField` và `minimumThresholdField` sang định dạng số (`Double/Integer`).
    - **Validation:** Cần logic kiểm tra kỹ lưỡng để đảm bảo người dùng không nhập chữ cái vào trường số hoặc nhập số âm.
- **Chỉ dấu về Logic Nghiệp vụ (Business Logic Indicators)** Trường `minimumThresholdField` (Ngưỡng tối thiểu) không chỉ là dữ liệu lưu trữ mà còn là tham số kích hoạt logic cảnh báo: Hệ thống sẽ dựa vào giá trị

này để tô màu đỏ hoặc gửi thông báo khi tồn kho xuống thấp ("Low Stock Alert").

e) Khai báo Field UI Components Stock Form

```
// Stock In/Out fields
private ComboBox<String> stockItemComboBox;
private TextField stockQuantityField;
private ComboBox<String> stockReasonComboBox;
private ComboBox<String> stockTypeComboBox;
```

Đoạn mã từ dòng 37 đến 41 định nghĩa form chức năng thứ hai của Controller, tập trung vào các **Thao tác Nghiệp vụ (Operational Workflows)** thay vì quản lý dữ liệu tĩnh.

- **Liên kết Dữ liệu (Data Association):** `stockItemComboBox` đóng vai trò then chốt trong việc tạo mối quan hệ giữa giao dịch và hàng hóa. Nó hoạt động như một "Foreign Key" trên giao diện, buộc người dùng phải chọn một mặt hàng hiện hữu từ danh sách `itemList` để thực hiện giao dịch, đảm bảo tính toàn vẹn tham chiếu.
- **Phân loại Giao dịch:** Sự hiện diện của `stockTypeComboBox` (IN/OUT) và `stockReasonComboBox` gợi ý rằng Controller sẽ chứa logic điều hướng luồng xử lý: cộng dồn số lượng nếu là "IN" và trừ đi nếu là "OUT", kèm theo việc ghi nhận lý do để phục vụ báo cáo kiểm kê.

Thiết kế này khẳng định Controller đang thực hiện mô hình **Dual-Context Management**: vừa quản lý danh mục hàng hóa (Master Data), vừa xử lý các biến động kho (Transaction Data) trên cùng một màn hình.

3.3.3.2 Constructor và khởi tạo

```
public InventoryController() {
    this.inventoryService = new InventoryService();
    this.itemList = FXCollections.observableArrayList();
    this.transactionList = FXCollections.observableArrayList();
    loadItems();
    loadTransactions();
}
```

Đoạn mã từ dòng 43 đến 48 thiết lập trạng thái ban đầu cho `InventoryController`, thể hiện rõ vai trò **Quản lý Hợp nhất (Unified Management)** của lớp này:

- **Khởi tạo Trạng thái Đa Chiều (Multi-Dimensional State Setup)** Việc khởi tạo đồng thời hai đối tượng `ObservableList` (`itemList` và `transactionList`) tại dòng 45-46 xác nhận rằng Controller này phải duy trì sự đồng bộ giữa hai miền dữ liệu:
  - **Inventory Snapshot:** Dữ liệu tĩnh về trạng thái kho hiện tại.
  - **Audit Trail:** Dữ liệu động về lịch sử dòng chảy vật tư.
- **Cơ chế Tải Dữ liệu Sớm (Eager Data Loading)** Khác với chiến lược "Lazy Loading" (chỉ tải khi người dùng yêu cầu), phương thức này áp dụng **Eager Loading** thông qua việc gọi ngay lập tức `loadItems()` và `loadTransactions()` (dòng 47-48). Điều này đảm bảo toàn bộ dữ liệu nghiệp vụ sẵn sàng hiển thị ngay khi View được render, giúp trải nghiệm người dùng mượt mà nhưng đòi hỏi Service phải xử lý hiệu quả để tránh làm chậm quá trình khởi động ứng dụng.

3.3.3.3 Method setItemTableView(): Ràng buộc Item TableView

```
    /**
     * Set the item TableView reference.
     */
    public    void    setItemTableView(TableView<InventoryItem>
tableView) {
        this.itemTableView = tableView;
        tableView.setItems(itemList);
    }
```

Đoạn mã từ dòng 51 đến 55 thực hiện bước **Dependency Injection** cho thành phần hiển thị quan trọng nhất: Bảng tồn kho.

- **Giải quyết sự nhập nhằng (Disambiguation):** Việc đặt tên phương thức cụ thể là `setItemTableView` (thay vì `setTableView` chung chung) là bắt buộc về mặt kiến trúc. Do Controller này quản lý đồng thời hai luồng dữ liệu,

việc định danh rõ ràng giúp `MainApp` phân biệt chính xác bảng nào hiển thị Hàng hóa và bảng nào hiển thị Giao dịch khi thực hiện binding.

– **Thiết lập Binding:** Dòng 55 `tableView.setItems(itemList)` kích hoạt cơ chế quan sát. Từ thời điểm này, mọi thay đổi trong danh sách `itemList` (Model) sẽ tự động phản ánh lên giao diện (View) mà không cần can thiệp thủ công.

3.3.3.4 Method setTransactionTableView(): Ràng buộc Transaction TableView

```
    /**
     * Set the transaction TableView reference.
     */
    public void setTransactionTableView(TableView<InventoryTransaction>
tableView) {
        this.transactionTableView = tableView;
        tableView.setItems(transactionList);
    }
```

Đoạn mã từ dòng 59 đến 63 hoàn thiện mô hình hiển thị kép của `InventoryController` bằng cách thiết lập **Bảng Lịch sử Giao dịch (Transaction Log)**.

– **Kiến trúc Đối xứng (Symmetrical Architecture):** Phương thức này là bản sao logic của `setItemTableView`, nhưng phục vụ cho thực thể `InventoryTransaction`. Sự tồn tại song song của hai phương thức setter riêng biệt (`setItem...` và `setTransaction...`) khẳng định tính độc lập của hai vùng hiển thị, cho phép `MainApp` điều phối dữ liệu vào đúng đích mà không gây xung đột.

– **Binding Dữ liệu Sự kiện:** Dòng 63 `tableView.setItems(transactionList)` kết nối dòng dữ liệu nhập/xuất với giao diện. Điều này rất quan trọng về mặt UX: người dùng sẽ thấy ngay dòng nhật ký vừa tạo xuất hiện trên bảng dưới sau khi thực hiện thao tác kho, tạo cảm giác phản hồi tức thì.

3.3.3.5 Method setItemFormFields(): Đăng ký Item Form Fields và Populate Combo Boxes

```
    /**
     * Set form input fields for item management.
     */
    public void setItemFormFields(TextField idField, TextField
nameField, ComboBox<String> categoryComboBox,
                                  ComboBox<String>  unitComboBox,
TextField quantityField,
                                  TextField minimumThresholdField,
TextField supplierNameField,
                                  ComboBox<String>
storageLocationComboBox) {
        this.idField = idField;
        this.nameField = nameField;
        this.categoryComboBox = categoryComboBox;
        this.unitComboBox = unitComboBox;
        this.quantityField = quantityField;
        this.minimumThresholdField = minimumThresholdField;
        this.supplierNameField = supplierNameField;
        this.storageLocationComboBox = storageLocationComboBox;

        // Populate combo boxes
        categoryComboBox.getItems().addAll("Food",    "Beverage",
"Ingredient", "Cleaning", "Others");
        unitComboBox.getItems().addAll("kg",  "liter",  "piece",
"box", "pack", "bottle", "can", "bag");
        storageLocationComboBox.getItems().addAll("Kitchen",
"Bar", "Warehouse", "Freezer");
    }
```

Đoạn mã này thực hiện quá trình **UI Binding** và **Data Seeding** cho form quản lý Inventory Item, đảm bảo các thành phần giao diện sẵn sàng tương tác ngay khi View được load.

- **Explicit UI Dependency Injection (Dòng 67-77)** Phương thức setItemFormFields đóng vai trò là một **Injector Method**, nhận tham chiếu của 8 UI Controls từ MainApp và gán vào các **Instance Fields** của Controller.
    - **Namespace Separation:** Prefix "Item" trong tên method (setItem...) là cần thiết để phân biệt scope với các method xử lý Stock Transaction, tuân thủ nguyên tắc **Naming Convention** rõ ràng trong một Controller quản lý đa chức năng.

- **State Management:** Việc lưu trữ reference cho phép Controller toàn quyền truy xuất (get value), thay đổi (set value) hoặc reset trạng thái của form trong suốt vòng đời ứng dụng.

– **Static Data Seeding & Domain Knowledge (Dòng 79-81)** Controller thực hiện **Pre-population** dữ liệu cho các ComboBox ngay tại thời điểm khởi tạo:

- **Hardcoded Values:** Các giá trị như "Food", "Beverage" (Category) hay "Kitchen", "Bar" (Location) được định nghĩa cứng (Hardcoded). Đây là các **Constant Domain Values** đặc thù của nghiệp vụ nhà hàng, hiếm khi thay đổi.
- **Trade-off:** Cách tiếp cận này giúp giảm tải **Database calls** (không cần bảng lookup riêng trong DB), tối ưu hiệu năng (Performance) nhưng đánh đổi bằng tính linh hoạt (Flexibility) – nếu muốn thêm Category mới, Developer phải sửa code thay vì Config.
- **UX Optimization:** Đảm bảo **Dropdown Lists** luôn có dữ liệu hợp lệ (Valid State) ngay khi người dùng mở form, tránh lỗi NullPointerException hoặc trải nghiệm "Empty State" không mong muốn.

3.3.3.6 Method setStockFormFields(): Đăng ký Stock Form Fields và Populate Combo Boxes

```
java
/**
 * Set form input fields for stock in/out.
 */
public void setStockFormFields(ComboBox<String>
stockItemComboBox, TextField stockQuantityField,
                               ComboBox<String>
stockReasonComboBox, ComboBox<String> stockTypeComboBox) {
    this.stockItemComboBox = stockItemComboBox;
    this.stockQuantityField = stockQuantityField;
    this.stockReasonComboBox = stockReasonComboBox;
    this.stockTypeComboBox = stockTypeComboBox;

    // Populate combo boxes
    stockReasonComboBox.getItems().addAll("Purchase",
"Sale", "Waste", "Adjustment", "Return");
```

```
        stockTypeComboBox.getItems().addAll("IN", "OUT");
        stockTypeComboBox.setValue("IN");

        // Populate item combo box
        refreshItemComboBox();
    }
```

Đoạn mã này thiết lập môi trường cho chức năng thứ hai của Controller: Quản lý biến động kho (Stock Flow). Điểm đáng chú ý là sự kết hợp giữa **Static Data** và **Dynamic Data** trong cùng một form.

- **Static Data Seeding & Smart Defaults (Dòng 97-99)** Controller tiếp tục áp dụng pattern hardcoding cho các dữ liệu mang tính cấu hình hệ thống:
  - **Business Logic Enforcement:** Các giá trị Reason (Purchase, Sale, Waste...) và Type (IN, OUT) định hình chặt chẽ quy trình nghiệp vụ, ngăn chặn người dùng nhập liệu tự do.
  - **UX Optimization (Smart Default):** Việc setValue("IN") giúp giảm thiểu thao tác (clicks) cho người dùng, dựa trên giả định rằng thao tác Nhập kho (Restocking) là hành vi khởi tạo phổ biến hoặc cần sự an toàn cao hơn.
- **Dynamic Data Binding & Dependency (Dòng 101-102)** Đây là điểm khác biệt kiến trúc so với form Item:
  - **Inter-component Communication:** Việc gọi refreshItemComboBox() tạo ra mối liên kết dữ liệu giữa module **Item Management** (nguồn dữ liệu) và module **Stock Operations** (nơi tiêu thụ dữ liệu).
  - **Real-time Updates:** stockItemComboBox không chứa dữ liệu tĩnh mà phụ thuộc vào trạng thái hiện tại của itemList. Điều này đảm bảo tính toàn vẹn tham chiếu: người dùng chỉ có thể tạo giao dịch cho các Item thực sự tồn tại trong hệ thống.

3.3.3.7 Method refreshItemComboBox(): Cập nhật Item Combo Box

```
    /**
     * Refresh item combo box for stock operations.
     */
    public void refreshItemComboBox() {
```

```
        if (stockItemComboBox == null) {
            return;
        }
        stockItemComboBox.getItems().clear();
        for (InventoryItem item : itemList) {
            stockItemComboBox.getItems().add(item.getId()  +
" " + item.getName());
        }
    }
```

Phương thức `refreshItemComboBox` đóng vai trò là cơ chế **State Synchronization** (Đồng bộ trạng thái) giữa hai module: Quản lý Danh mục (Master Data) và Tác nghiệp Kho (Transaction).

- **Defensive Programming (Dòng 111-113)** Kỹ thuật **Null Safety** được áp dụng chặt chẽ. Do phương thức này có thể được gọi từ nhiều ngữ cảnh (khi load data, khi thêm/sửa item), việc kiểm tra `if (stockItemComboBox == null)` là bắt buộc để ngăn chặn **Race Conditions** trong quá trình khởi tạo UI (UI Initialization Lifecycle), đảm bảo ứng dụng không bị crash do `NullPointerException`.
- **Data Transformation & Formatting (Dòng 115-117)** Logic này thực hiện một bước chuyển đổi dữ liệu đơn giản (View Model Transformation) từ Object sang String:
    - **Format Pattern:** Chuỗi "ID Name" (ví dụ: "I001 Rice") cung cấp ngữ cảnh đầy đủ cho người dùng.
    - **User Experience (UX):** Giúp người dùng phân biệt chính xác các mặt hàng có tên giống nhau (nhờ ID) hoặc tìm kiếm nhanh nếu chỉ nhớ tên, giải quyết vấn đề **Ambiguity** trong chọn lựa.
    - **Dynamic Population Pattern** Khác với các ComboBox "tĩnh" (Category, Unit), `stockItemComboBox` áp dụng **Dynamic Population**. Nó phản ánh thời gian thực trạng thái của `itemList`. Bất kỳ thay đổi nào ở module Item (CRUD operations) đều cần trigger phương thức này để đảm bảo dữ liệu trong form Stock luôn là dữ liệu mới nhất (Fresh Data).

3.3.3.8 Method loadItems(): Tải Items từ Service

```
    /**
     * Load all items into the table.
     */
    public void loadItems() {
        itemList.clear();
        itemList.addAll(inventoryService.getAllItems());
        refreshItemComboBox();
    }
```

Đoạn mã từ dòng 121 đến 126 đóng vai trò cốt lõi trong cơ chế **Data Binding** và **State Management** của Controller, đảm bảo tính nhất quán giữa dữ liệu nghiệp vụ và giao diện người dùng (UI).

Về mặt cấu trúc, các dòng 121-123 duy trì tiêu chuẩn coding convention của dự án thông qua **JavaDoc** và **method signature** tương đồng với các Controller khác, tạo sự thống nhất trong kiến trúc tổng thể. Tuy nhiên, logic xử lý chính nằm ở quy trình làm mới dữ liệu (Data Refresh Workflow) từ dòng 124 đến 126:

- **Cơ chế làm sạch và đồng bộ (Dòng 124-125):** Quy trình bắt đầu bằng việc gọi itemList.clear() tại dòng 124 để giải phóng bộ nhớ đệm cục bộ (local collection), ngăn chặn tình trạng trùng lặp dữ liệu (data duplication). Ngay sau đó, dòng 125 thực thi itemList.addAll(inventoryService.getAllItems()), thực hiện việc fetch dữ liệu mới nhất từ **Service Layer**. Bước này đảm bảo danh sách itemList trong Controller luôn phản ánh chính xác trạng thái hiện tại của Database ("Single Source of Truth").
- **Tự động cập nhật giao diện (Dòng 126):** Điểm khác biệt chiến lược nằm ở dòng 126 với phương thức refreshItemComboBox(). Việc kích hoạt phương thức này ngay sau khi tải dữ liệu thiết lập một cơ chế **Auto-refresh** chặt chẽ. Cụ thể, nó buộc Stock Form ComboBox phải render lại dựa trên dữ liệu mới, đảm bảo tính đồng bộ (Synchronization) tuyệt đối giữa module **Item Management** và các nghiệp vụ **Stock Operations**.

Tổng quan về Lifecycle: Phương thức này đóng vai trò như một **Synchronization Hub** trong vòng đời ứng dụng (Application Lifecycle). Nó được thiết kế để trigger trong ba ngữ cảnh quan trọng:

- Giai đoạn khởi tạo (**Constructor**) để nạp dữ liệu ban đầu.
- Sau các tác vụ **CRUD** (Create, Read, Update, Delete) để làm mới View.
- Sau các **Stock Operations** để cập nhật thuộc tính quantity của items theo thời gian thực.

3.3.3.9 Method loadTransactions(): Tải Transactions từ Service

```
    /**
     * Load all transactions into the table.
     */
    public void loadTransactions() {
        transactionList.clear();
        transactionList.addAll(inventoryService.getAllTransactions());
    }
```

Đoạn mã từ dòng 130 đến 134 thể hiện việc áp dụng nguyên tắc **Separation of Concerns (SoC)** trong thiết kế Controller, cụ thể là sự tách biệt giữa luồng dữ liệu (Data Stream) của Transactions và Items.

Về mặt cấu trúc, các dòng 130-132 định nghĩa một method signature riêng biệt cùng hệ thống JavaDoc chuẩn mực. Việc tách logic tải dữ liệu này ra khỏi `loadItems()` cho thấy Controller được thiết kế để quản lý hai thực thể dữ liệu (**Entities**) độc lập với các vòng đời (**Lifecycles**) riêng biệt. Điều này giúp tăng cường tính module hóa (**Modularity**) và khả năng bảo trì của mã nguồn.

Quy trình đồng bộ hóa dữ liệu được thực hiện qua hai bước tại dòng 133 và 134:

- **State Reset (Dòng 133):** Phương thức `transactionList.clear()` được gọi để làm sạch **Collection** cục bộ, đảm bảo không xảy ra xung đột hoặc trùng lặp dữ liệu trước khi nạp mới.
- **Data Synchronization (Dòng 134):**

`transactionList.addAll(inventoryService.getAllTransactions())` thực hiện việc fetch toàn bộ lịch sử giao dịch từ **Service Layer**, đảm bảo dữ liệu hiển thị trên View luôn nhất quán với Database.

Ngữ cảnh thực thi (Execution Context): Phương thức này đóng vai trò quan trọng trong việc duy trì **Data Integrity** (tính toàn vẹn dữ liệu) xuyên suốt vòng đời ứng dụng:

- Được kích hoạt trong **Constructor** để khởi tạo trạng thái ban đầu (Initialization).
- Được gọi lại (Callback) sau các **Stock Operations** để cập nhật nhật ký giao dịch mới nhất.
- Xử lý logic hậu kỳ (Post-processing) sau khi xóa Item để xử lý các ràng buộc toàn vẹn (ví dụ: **Cascading Delete** các transactions liên quan).

3.3.3.10 Method handleAdd(): Xử lý Thêm Item

```
    /**
     * Handle add button click.
     */
    public void handleAdd() {
        try {
            InventoryItem item = createItemFromForm();
            inventoryService.addItem(item);
            loadItems();
            clearForm();
            showSuccessAlert("Inventory item added successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error adding item", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 138 đến 148 hiện thực hóa chức năng **Create** trong chu trình CRUD, đóng vai trò là điểm tương tác chính (Entry Point) giữa người dùng và hệ thống quản lý kho. Cấu trúc logic được thiết kế để đảm bảo tính toàn vẹn dữ liệu (**Data Integrity**) và trải nghiệm người dùng (**User Experience UX**) liền mạch.

1. **Cơ chế Đóng gói và Xử lý Lỗi (Exception Handling & Encapsulation)**

- **Tiêu chuẩn hóa (Dòng 138-140):** Việc tuân thủ JavaDoc và method signature chuẩn giúp duy trì tính nhất quán (**Consistency**) trong kiến trúc Controller, hỗ trợ khả năng đọc hiểu và bảo trì mã nguồn.
- **Quản lý ngoại lệ (Dòng 141, 147-148):** Toàn bộ logic nghiệp vụ được bao bọc trong khối try-catch. Đây là cơ chế phòng vệ thiết yếu để bắt các lỗi Runtime (như lỗi kết nối Database hoặc Validation), ngăn chặn ứng dụng bị crash và cung cấp phản hồi lỗi phù hợp thông qua cơ chế logging hoặc alert.

2. **Quy trình Nghiệp vụ Cốt lõi (Core Business Logic)**

Luồng xử lý dữ liệu được thực hiện tuần tự qua các bước sau:

- **Data Binding & Transformation (Dòng 142):** Phương thức createItemFromForm() hoạt động như một **Helper Method**, chịu trách nhiệm thu thập dữ liệu thô (raw data) từ các UI controls và đóng gói (**Encapsulate**) chúng thành một đối tượng InventoryItem hoàn chỉnh. Bước này tách biệt logic giao diện khỏi logic nghiệp vụ.
- **Service Delegation (Dòng 143):** inventoryService.addItem(item) thực hiện việc ủy quyền xử lý xuống **Service Layer**. Tại đây, các quy tắc nghiệp vụ (Business Rules) và thao tác persistence vào Database sẽ được thực thi.

3. **Đồng bộ hóa Trạng thái và Tích hợp (State Synchronization & Integration)**

- **Cập nhật Giao diện (Dòng 144):** Việc gọi loadItems() ngay sau khi thêm mới thành công là bước quan trọng nhất trong khối lệnh này. Nó không chỉ làm mới danh sách hiển thị trên bảng chính mà còn kích hoạt cơ chế **Auto-refresh** cho Stock Form ComboBox (như đã phân tích ở dòng 126).
    - *Ý nghĩa kiến trúc:* Điều này thể hiện sự tích hợp chặt chẽ (**Tight Integration**) giữa module quản lý Item và module Stock, đảm bảo rằng ngay khi một Item được tạo, nó lập tức khả dụng cho các thao tác nhập/xuất kho mà không cần người dùng thao tác lại.
- **Phản hồi người dùng (Dòng 145-146):** Các phương thức clearForm() và showSuccessAlert() hoàn tất chu trình tương tác bằng việc reset trạng thái

nhập liệu và xác nhận thành công, tuân thủ nguyên tắc thiết kế giao diện hướng người dùng.

3.3.3.11 Method handleUpdate(): Xử lý Cập nhật Item

```
    /**
     * Handle update button click.
     */
    public void handleUpdate() {
        try {
            InventoryItem item = createItemFromForm();
            if (item.getId() == null || item.getId().isEmpty()) {
                showErrorAlert("Error", "Please select an item to
update");
                return;
            }
            inventoryService.updateItem(item);
            loadItems();
            clearForm();
            showSuccessAlert("Inventory item updated successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error updating item", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 153 đến 167 hiện thực hóa chức năng **Update** trong mô hình CRUD. Đây là quy trình nghiệp vụ quan trọng nhằm duy trì tính chính xác của dữ liệu tồn kho theo thời gian. Mặc dù cấu trúc tương đồng với `EmployeeController`, việc áp dụng logic này vào `InventoryController` mang những đặc thù riêng về tính toàn vẹn dữ liệu.

1) **Chuẩn hóa Cấu trúc và Xử lý Lỗi (Standardization & Exception Handling)**

– **Method Signature (Dòng 153-155):** Việc duy trì JavaDoc và định nghĩa phương thức thống nhất minh chứng cho việc áp dụng Coding Standard chặt chẽ trên toàn dự án, giúp giảm thiểu cognitive load (tải nhận thức) cho lập trình viên khi bảo trì mã nguồn.

– **Robustness (Dòng 156, 166-167):** Khối `try-catch` bao bọc toàn bộ logic nghiệp vụ đóng vai trò như một Safety Net, đảm bảo ứng dụng có khả năng chịu lỗi (Fault Tolerance) và ngăn chặn việc crash chương trình khi gặp ngoại lệ runtime.

2) **Validation và Ràng buộc Dữ liệu (Dòng 157-161)**

Quy trình này đảm bảo tính hợp lệ của dữ liệu trước khi tác động vào Database:

- Object Mapping (Dòng 157): `createItemFromForm()` thực hiện việc map dữ liệu từ UI Form sang đối tượng Domain (`InventoryItem`).
- Identity Validation (Dòng 158-161): Đây là bước kiểm tra logic quan trọng nhất ("Guard Clause"). Hệ thống kiểm tra tính tồn tại của Primary Key (ID).
    - Nếu ID là `null` hoặc rỗng, điều này ám chỉ người dùng đang cố gắng "Update" một đối tượng chưa được định danh (chưa chọn từ danh sách).
    - Hệ thống từ chối thao tác và trả về thông báo lỗi, ngăn chặn các hành vi cập nhật không xác định (Undefined Behavior).

3) **Tương tác Service và Đồng bộ hóa (Dòng 162-165)**

- Service Invocation (Dòng 162): `inventoryService.updateItem(item)` ủy quyền cho Service Layer thực hiện logic cập nhật xuống tầng Persistence.
- State Synchronization (Dòng 163): Lệnh `loadItems()` tại đây có tác động kép:
    - Cập nhật `TableView` chính để phản ánh thông tin mới sửa.
    - Quan trọng hơn, nó kích hoạt `refreshItemComboBox()`, đảm bảo Stock Form ngay lập tức nhận diện được sự thay đổi của Item (ví dụ: nếu tên Item bị sửa, Dropdown menu nhập hàng cũng phải hiển thị tên mới này).
- UI Reset (Dòng 164-165): Làm sạch Form và thông báo thành công để kết thúc transaction UX.

4) **Tính nhất quán kiến trúc (Architectural Consistency)**

Việc phương thức này có cấu trúc giống hệt `EmployeeController.handleUpdate()` thể hiện việc áp dụng Design Pattern (có thể coi là một dạng *Template Method Pattern* ngầm định). Việc tái sử dụng cấu trúc logic cho các Entity khác nhau (Item vs Employee) giúp hệ thống dễ

mở rộng và kiểm thử (Testability), vì hành vi của các Controller là dự đoán được (Predictable).

3.3.3.12 Method handleDelete(): Xử lý Xóa Item

```
    /**
     * Handle delete button click.
     */
    public void handleDelete() {
        InventoryItem selected = itemTableView.getSelectionModel().getSelectedItem();
        if (selected == null) {
            showErrorAlert("Error", "Please select an item to delete");
            return;
        }

        try {
            inventoryService.deleteItem(selected.getId());
            loadItems();
            loadTransactions();
            clearForm();
            showSuccessAlert("Inventory item deleted successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error deleting item", e.getMessage());
        }
    }
```

Đoạn mã này hiện thực hóa thao tác **Delete**, một bước quan trọng trong chu trình quản lý dữ liệu, nơi tính toàn vẹn tham chiếu (**Referential Integrity**) được đặt lên hàng đầu.

- **Kiểm soát Tiền điều kiện (Pre-condition Check):** Tại dòng 175-179, hệ thống thực hiện validation đối tượng được chọn từ giao diện. Logic này đóng vai trò như một **Guard Clause**, ngăn chặn việc thực thi các lệnh gọi service không hợp lệ (Null Reference) khi người dùng chưa chọn đối tượng mục tiêu.
- **Xử lý Nghiệp vụ và Lan truyền Trạng thái (Propagation):** Trọng tâm logic nằm ở dòng 181-187. Sau khi `inventoryService.deleteItem()` được thực thi thành công:
    - **Primary Update:** `loadItems()` được gọi để loại bỏ item khỏi danh sách hiển thị và ComboBox.
    - **Secondary Update (Điểm khác biệt cốt lõi):** Việc gọi thêm `loadTransactions()` thể hiện sự phụ thuộc dữ liệu (**Data**

Dependency) chặt chẽ giữa `Item` và `Transaction`. Trong mô hình dữ liệu quan hệ, việc xóa một Item thường kích hoạt cơ chế **Cascade Delete** hoặc cập nhật trạng thái các Transaction liên quan. Controller bắt buộc phải refresh cả danh sách Transaction để đảm bảo UI không hiển thị dữ liệu "mồ côi" (Orphaned Data).

- **So sánh Kiến trúc:** Khác với `EmployeeController`, phương thức này phải quản lý sự đồng bộ của hai tập dữ liệu song song. Điều này minh chứng rằng `InventoryController` không chỉ đơn thuần quản lý một thực thể (Entity) mà còn điều phối mối quan hệ phức tạp giữa Kho hàng và Lịch sử giao dịch.

3.3.3.13 Method handleTableSelection(): Xử lý Lựa chọn Hàng trong Bảng

```
    /**
     * Handle table row selection.
     */
    public void handleTableSelection() {
        InventoryItem selected = itemTableView.getSelectionModel().getSelectedItem();
        if (selected != null) {
            populateForm(selected);
        }
    }
```

Đoạn mã từ dòng 193 đến 199 thực hiện cơ chế **Data Binding** một chiều từ bảng danh sách lên giao diện nhập liệu. Tại dòng 196, hệ thống truy xuất đối tượng hiện hành thông qua itemTableView.getSelectionModel().getSelectedItem(). Khối lệnh điều kiện từ dòng 197 đến 199 đảm bảo rằng khi một đối tượng hợp lệ được chọn, phương thức populateForm(selected) sẽ được kích hoạt để ánh xạ dữ liệu vào các trường UI, chuẩn bị ngữ cảnh cho thao tác cập nhật. Phương thức này được thiết kế chuyên biệt cho itemTableView nhằm xử lý các thực thể có thể chỉnh sửa (**Mutable Entities**). Ngược lại, transactionTableView bị loại trừ khỏi luồng xử lý này, khẳng định vai trò **Read-only** (chỉ đọc) của dữ liệu lịch sử giao dịch trong hệ thống.

3.3.3.14 Method handleStockOperation(): Xử lý Nhập/Xuất Kho

```
    /**
     * Handle stock in/out button click.
     */
    public void handleStockOperation(String staffId, String
staffName) {
        try {
            String itemSelection = stockItemComboBox.getValue();
            if (itemSelection == null || itemSelection.isEmpty())
{
                showErrorAlert("Error", "Please select an item");
                return;
            }

            String itemId = itemSelection.split(" ")[0];
            double quantity =
Double.parseDouble(stockQuantityField.getText().trim());
            String reason = stockReasonComboBox.getValue();
            String type = stockTypeComboBox.getValue();

            if (reason == null || reason.isEmpty()) {
                showErrorAlert("Error", "Please select a reason");
                return;
            }

            if (type == null || type.isEmpty()) {
                showErrorAlert("Error", "Please select operation
type (IN/OUT)");
                return;
            }

            InventoryTransaction transaction;
            if ("IN".equals(type)) {
                transaction = inventoryService.stockIn(itemId,
quantity, reason, staffId, staffName);
            } else {
                transaction = inventoryService.stockOut(itemId,
quantity, reason, staffId, staffName);
            }

            loadItems();
            loadTransactions();
            clearStockForm();
            showSuccessAlert("Stock " + type + " operation
completed successfully!");
        } catch (NumberFormatException e) {
            showErrorAlert("Error", "Please enter a valid
quantity");
```

```
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 203 đến 244 hiện thực hóa quy trình nghiệp vụ phức tạp nhất trong Controller: xử lý giao dịch Nhập/Xuất kho (**Stock In/Out**) với sự tích hợp đa chiều giữa dữ liệu và giao diện.

- **Định danh và Thiết lập Context (Dòng 203-205):** Method signature tiếp nhận tham số `staffId` và `staffName` nhằm phục vụ mục đích **Audit Trail** (lưu vết người thực hiện). Dù hiện tại được hardcode ("SYS"), về mặt thiết kế, đây là điểm chờ (hook) để tích hợp với **Session Management** trong môi trường Production.
- **Validation và Parsing Dữ liệu (Dòng 206-227):** Toàn bộ logic được bao bọc trong khối `try-catch` để đảm bảo tính ổn định (**Robustness**). Trước khi xử lý, hệ thống thực hiện chuỗi các **Guard Clauses** (dòng 207-227) để kiểm tra tính hợp lệ của input:
    - **Data Parsing:** Sử dụng kỹ thuật `String Manipulation` (split) để trích xuất Item ID từ chuỗi định dạng UI của ComboBox.
    - **Type Casting:** Sử dụng `Double.parseDouble` (dòng 214) thay vì Integer, cho phép xử lý các đơn vị đo lường thập phân, đồng thời bắt lỗi định dạng thông qua `NumberFormatException`.
- **Ủy quyền Service và Điều hướng Logic (Dòng 229-234):** Dựa trên `Transaction Type` (IN hoặc OUT), Controller thực hiện **Service Delegation** bằng cách gọi phương thức tương ứng (`stockIn` hoặc `stockOut`). Tầng Service chịu trách nhiệm đảm bảo tính toàn vẹn dữ liệu: cập nhật số lượng tồn kho của Item và khởi tạo bản ghi `InventoryTransaction`.
- **Đồng bộ Trạng thái và Xử lý Lỗi (Dòng 236-244):** Sau khi giao dịch thành công, cơ chế **Dual Refresh** được kích hoạt: gọi `loadItems()` để cập nhật số lượng tồn mới nhất và `loadTransactions()` để hiển thị lịch sử giao

dịch vừa tạo, đảm bảo tính nhất quán (**Data Consistency**) trên toàn bộ View. Cuối cùng, khối `catch` xử lý các ngoại lệ đặc thù: `NumberFormatException` cho lỗi nhập liệu và `IllegalArgumentException` cho lỗi nghiệp vụ (ví dụ: xuất quá số lượng tồn), cung cấp phản hồi chính xác cho người dùng.

3.3.3.15 Method createItemFromForm(): Tạo InventoryItem Object từ Form

```
    /**
     * Create InventoryItem object from form fields.
     */
    private InventoryItem createItemFromForm() {
        InventoryItem item = new InventoryItem();
        item.setId(idField.getText().trim());
        item.setName(nameField.getText().trim());
        item.setCategory(categoryComboBox.getValue());
        item.setUnit(unitComboBox.getValue());

        try {
            item.setQuantity(Double.parseDouble(quantityField.getText()
.trim()));
            item.setMinimumThreshold(Double.parseDouble(minimumThreshol
dField.getText().trim()));
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Quantity and threshold
must be valid numbers");
        }

        item.setSupplierName(supplierNameField.getText().trim());
        item.setStorageLocation(storageLocationComboBox.getValue());

        return item;
    }
```

Đoạn mã từ dòng 247 đến 266 hiện thực hóa một **Private Helper Method**, chịu trách nhiệm chuyển đổi dữ liệu thô từ giao diện (UI) thành đối tượng nghiệp vụ (**Domain Object**).

- **Khởi tạo và Binding Dữ liệu (Dòng 250-254, 263-264):** Phương thức bắt đầu bằng việc khởi tạo instance `InventoryItem` (dòng 250). Tiếp theo, quá trình **Data Binding** được thực hiện thủ công để ánh xạ các giá trị từ TextFields (ID, Name) và ComboBoxes (Category, Unit, Supplier, Location) vào các thuộc tính tương ứng của đối tượng.

- **Xử lý Kiểu số và Exception Translation (Dòng 256-261):** Đây là logic quan trọng nhất phân biệt phương thức này với `EmployeeController`. Hệ thống thực hiện **Data Transformation** ép kiểu chuỗi sang `Double` cho các trường Quantity và Threshold.
    - **Decimal Support:** Sử dụng `Double` thay vì `Integer` để hỗ trợ các đơn vị đo lường thập phân chính xác.
    - **Exception Wrapping:** Khối `try-catch` tại đây thực hiện kỹ thuật **Exception Translation**. Nó bắt lỗi cấp thấp `NumberFormatException` (lỗi cú pháp) và bọc lại thành `IllegalArgumentException` (lỗi nghiệp vụ) với thông điệp rõ ràng hơn ("Quantity and Threshold must be valid numbers"). Điều này giúp tách biệt tầng giao diện khỏi tầng xử lý lỗi logic.
- **Đánh giá Độ phức tạp:** So với `EmployeeController.createEmployeeFromForm()`, phương thức này có độ phức tạp cao hơn (Higher Complexity) do phải xử lý validation chặt chẽ cho các kiểu dữ liệu số thực và quản lý trạng thái của nhiều **UI Controls** (Multiple ComboBoxes), đảm bảo đối tượng trả về tại dòng 266 luôn ở trạng thái hợp lệ (**Valid State**) trước khi chuyển xuống Service Layer.

3.3.3.16 Method populateForm(): Điền Form từ InventoryItem Object

```
    /**
     * Populate form fields with selected item data.
     */
    private void populateForm(InventoryItem item) {
        idField.setText(item.getId());
        nameField.setText(item.getName());
        categoryComboBox.setValue(item.getCategory());
        unitComboBox.setValue(item.getCategory());
        quantityField.setText(String.valueOf(item.getQuantity()));
        minimumThresholdField.setText(String.valueOf(item.getMinimumThr
eshold()));
        supplierNameField.setText(item.getSupplierName());
        storageLocationComboBox.setValue(item.getStorageLocation());
    }
```

Đoạn mã từ dòng 270 đến 281 hiện thực hóa quy trình **Data Binding** theo chiều từ Model sang View (Model-to-View), phục vụ cho chức năng xem chi tiết hoặc chỉnh sửa.

- **Cơ chế Đồng bộ UI (UI Synchronization):** Phương thức `private` này thực hiện việc "đổ" dữ liệu từ đối tượng `InventoryItem` vào các thành phần giao diện. Quá trình này đảm bảo trạng thái của Form phản ánh chính xác trạng thái của đối tượng đang được chọn (**Selection State**).
- **Xử lý Đa dạng Kiểu dữ liệu (Heterogeneous Data Handling):** Khác với các phương thức ánh xạ đơn giản, logic tại đây xử lý ba loại dữ liệu khác nhau:
  - **Direct Mapping:** Sử dụng `setText()` cho các trường chuỗi (String).
  - **Component State:** Sử dụng `setValue()` để thiết lập trạng thái cho các `ComboBox`.
  - **Type Conversion (Dòng 280-281):** Thực hiện ép kiểu tường minh từ `Double` sang `String` bằng phương thức `String.valueOf()`. Đây là bước bắt buộc để hiển thị dữ liệu số (Quantity, Threshold) lên các điều khiển văn bản (TextField).
- **So sánh độ phức tạp:** So với `EmployeeController.populateForm()`, phương thức này thể hiện độ phức tạp cao hơn do phải quản lý việc chuyển đổi định dạng số thực và đồng bộ trạng thái của nhiều danh sách chọn (ComboBoxes), đảm bảo tính chính xác về mặt hiển thị dữ liệu.

3.3.3.17 Method clearForm(): Xóa Item Form Fields

```
    /**
     * Clear form fields.
     */
    private void clearForm() {
        idField.clear();
        nameField.clear();
        categoryComboBox.setValue(null);
        unitComboBox.setValue(null);
        quantityField.clear();
        minimumThresholdField.clear();
```

```
        supplierNameField.clear();
        storageLocationComboBox.setValue(null);
        if (itemTableView != null) {
            itemTableView.getSelectionModel().clearSelection();
        }
    }
```

Đoạn mã từ dòng 284 đến 298 hiện thực hóa quy trình **UI State Reset** cho phân hệ quản lý Item, đảm bảo giao diện quay về trạng thái mặc định (**Default State**) sau các thao tác nghiệp vụ.

- **Làm sạch Component (Component Reset Dòng 287-295):** Phương thức thực hiện việc "xả" dữ liệu trên các điều khiển nhập liệu:
  - Gọi `clear()` đối với các `TextField` để xóa bộ đệm chuỗi.
  - Thiết lập giá trị `null` cho các `ComboBox` để hủy bỏ lựa chọn hiện hành (**Nullification**).
- **Quản lý Trạng thái Chọn (Selection State Dòng 296-298):** Hệ thống can thiệp vào `SelectionModel` của `itemTableView` để hủy bỏ dòng đang chọn. Điều này ngăn chặn các thao tác nhầm lẫn (**Unintended Operations**) lên đối tượng cũ khi người dùng bắt đầu một ngữ cảnh nhập liệu mới.
- **Nguyên tắc Thiết kế:** Việc tách biệt phương thức này với `clearStockForm()` thể hiện sự tuân thủ nguyên tắc **Separation of Concerns (SoC)**. Mặc dù cả hai đều là thao tác làm sạch form, nhưng chúng phục vụ hai ngữ cảnh nghiệp vụ (Item Management vs. Stock Operations) hoàn toàn độc lập và có vòng đời khác nhau.

3.3.3.18 Method clearStockForm(): Xóa Stock Form Fields

```
    /**
     * Clear stock form fields.
     */
    private void clearStockForm() {
        stockItemComboBox.setValue(null);
        stockQuantityField.clear();
        stockReasonComboBox.setValue(null);
        stockTypeComboBox.setValue("IN");
    }
```

Đoạn mã từ dòng 301 đến 308 hiện thực hóa quy trình **State Reset** riêng biệt cho phân hệ Stock Operations, đảm bảo tính độc lập với form quản lý Item.

- **Khởi tạo lại Trạng thái (Re-initialization):** Phương thức `private` này đưa các thành phần giao diện của Stock Form về trạng thái ban đầu. Các trường nhập liệu và chọn lựa thông thường được gán giá trị `null` hoặc rỗng để làm sạch ngữ cảnh nhập liệu.
- **Chiến lược Giá trị Mặc định (Default Value Strategy):** Điểm nhấn kỹ thuật nằm ở việc thiết lập `stockTypeComboBox.setValue("IN")`. Thay vì đưa về `null`, hệ thống áp dụng **Default State** là "IN" (Nhập kho).
    - **Tối ưu hóa UX:** Đây là quyết định thiết kế dựa trên hành vi người dùng (**User Behavior**), giả định rằng thao tác nhập kho xảy ra thường xuyên hơn hoặc là thao tác ưu tiên, giúp giảm thiểu số lần nhấp chuột (**Click Reduction**).
- **Kiến trúc Đa ngữ cảnh (Multi-context Architecture):** Sự tồn tại song song của `clearStockForm()` và `clearForm()` (dòng 284) khẳng định Controller đang quản lý hai **Operational Contexts** (ngữ cảnh vận hành) riêng biệt. Mỗi form sở hữu **Lifecycle** độc lập, cho phép người dùng thao tác trên form kho mà không ảnh hưởng đến dữ liệu đang nhập dở trên form item và ngược lại.

3.3.3.19 Method filterItems(): Lọc Items theo Tiêu chí

```
    /**
     * Filter items by search criteria.
     */
    public FilteredList<InventoryItem> filterItems(String
searchText, String categoryFilter, boolean lowStockOnly) {
        FilteredList<InventoryItem> filtered = new
FilteredList<>(itemList);

        filtered.setPredicate(item -> {
            boolean matchesSearch = searchText == null ||
searchText.isEmpty() ||
                    item.getName().toLowerCase().contains(searchT
ext.toLowerCase()) ||
```

```
                    item.getId().toLowerCase().contains(searchTex
t.toLowerCase());

            boolean matchesCategory = categoryFilter == null ||
categoryFilter.isEmpty() ||
                    item.getCategory().equals(categoryFilter);

            boolean    matchesLowStock    =    !lowStockOnly    ||
item.isLowStock();

            return    matchesSearch    &&    matchesCategory    &&
matchesLowStock;
        });

        return filtered;
    }
```

Đoạn mã từ dòng 311 đến 328 hiện thực hóa pattern **Functional Filtering** sử dụng sức mạnh của JavaFX Collections Framework để xử lý truy vấn dữ liệu trên giao diện.

- **Cơ chế Wrapper và Reactive Binding (Dòng 314):** Thay vì tạo ra một `ArrayList` tĩnh mới, phương thức khởi tạo một `FilteredList` bao bọc lấy `ObservableList` gốc (`itemList`).
    - **Dynamic View:** `FilteredList` hoạt động như một lớp view động. Bất kỳ thay đổi nào (thêm/sửa/xóa) trên danh sách gốc sẽ tự động được phản ánh và áp dụng lại bộ lọc ngay lập tức mà không cần gọi lại hàm lọc thủ công.
- **Predicate Logic và Lambda Expression (Dòng 316-326):** Trọng tâm logic nằm ở việc định nghĩa **Predicate** (hàm điều kiện trả về boolean) thông qua một biểu thức Lambda. Thuật toán lọc kết hợp ba điều kiện độc lập bằng phép toán logic **AND** (Conjunction):
    - **Text Search (Dòng 317-319):** Thực hiện so khớp chuỗi con (substring matching) không phân biệt hoa thường (**Case-insensitive**) trên cả hai trường định danh (ID) và hiển thị (Name).
    - **Categorization (Dòng 321-322):** Kiểm tra sự tương đồng chính xác của thuộc tính Category.

- **State-based Filtering (Dòng 324):** Kiểm tra trạng thái nghiệp vụ `isLowStock()`. Đây là logic lọc dựa trên trạng thái tính toán của thực thể.

– **Kết quả (Return Value):** Tại dòng 326, sự kết hợp của `matchesSearch && matchesCategory && matchesLowStock` đảm bảo một Item chỉ được render lên giao diện khi thỏa mãn **tất cả** các tiêu chí được kích hoạt. Phương thức trả về đối tượng `FilteredList` (dòng 328) để bind trực tiếp vào `TableView`, hoàn tất chu trình phản hồi giao diện thời gian thực.

3.3.3.20 Method showSuccessAlert(): Hiển thị Thông báo Thành công

```
    /**
     * Show success alert.
     */
    private void showSuccessAlert(String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Success");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 333 đến 341 hiện thực hóa cơ chế **User Feedback** thông qua việc hiển thị thông báo thành công (Success Alert).

– **Chuẩn hóa Giao diện (Standardization):** Method signature và JavaDoc tuân thủ nghiêm ngặt **Coding Conventions** của dự án, đảm bảo sự đồng nhất về mặt tài liệu kỹ thuật. Implementation (dòng 336-341) sử dụng lớp `Alert` của JavaFX để cung cấp phản hồi trực quan ngay sau khi một Transaction hoàn tất.

– **Đánh giá Kiến trúc và Refactoring:** Việc logic này xuất hiện y hệt trong `BookingController` và `EmployeeController` cho thấy sự vi phạm nguyên tắc **DRY (Don't Repeat Yourself)**.

- **Vấn đề:** Đây là mã lặp (**Boilerplate Code**) hay còn gọi là **Code Duplication**. Việc duy trì cùng một logic hiển thị thông báo ở nhiều nơi làm tăng chi phí bảo trì (Maintenance Cost).
- **Giải pháp Kiến trúc:** Đoạn mã này là ứng cử viên sáng giá cho việc **Refactoring**. Logic này nên được tách (extract) ra một **Utility Class** tĩnh (ví dụ: `AlertUtils.showSuccess()`) hoặc đưa vào một lớp cha (**BaseController**) để các Controller con kế thừa, giúp tối ưu hóa cấu trúc dự án.

3.3.3.21 Method showErrorAlert(): Hiển thị Thông báo Lỗi

```
    /**
     * Show error alert.
     */
    private void showErrorAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 344 đến 352 thực hiện chức năng hiển thị thông báo lỗi với cấu trúc và logic hoàn toàn trùng khớp các controller khác. Cụ thể, từ phần tài liệu JavaDoc và method signature (dòng 344-346) cho đến phần implementation bên trong (dòng 347-352) đều được giữ nguyên bản, không có sự khác biệt nào so với các phần khác của hệ thống.

3.3.3.22 Method getItemList(): Getter cho Item Observable List

```
    /**
     * Get observable list for binding.
     */
    public ObservableList<InventoryItem> getItemList() {
        return itemList;
    }
```

Đoạn mã từ dòng 355 đến 358 định nghĩa một phương thức **Getter public** tiêu chuẩn kèm theo JavaDoc, cho phép các thành phần bên ngoài truy cập vào `itemList`. Tại dòng 358, phương thức trả về tham chiếu trực tiếp đến danh sách

này. Trong cấu trúc ứng dụng, phương thức được `MainApp` gọi để thiết lập cơ chế **Data Binding**, liên kết dữ liệu từ Controller vào `TableView` hiển thị items.

3.3.3.23 Method getTransactionList(): Getter cho Transaction Observable List

```
    /**
     * Get observable list for transactions.
     */
    public ObservableList<InventoryTransaction> getTransactionList() {
        return transactionList;
    }
```

Đoạn mã từ dòng 362 đến 365 thiết lập cơ chế truy cập dữ liệu thông qua một `public` getter chuyên biệt, được định nghĩa bởi JavaDoc và method signature chuẩn. Tại dòng 365, phương thức trả về tham chiếu trực tiếp đến `transactionList`. Về mặt tích hợp, phương thức này được `MainApp` gọi để thực hiện binding dữ liệu vào `TableView` hiển thị giao dịch. Điểm đặc thù về mặt kiến trúc của Controller này là sự tồn tại của hai getter methods tách biệt, cho phép quản lý và truy xuất hai danh sách dữ liệu (Items và Transactions) hoàn toàn độc lập.

3.3.3.24 Tổng kết

`InventoryController` được xác định là thành phần phức tạp nhất trong kiến trúc hệ thống, đóng vai trò cầu nối trọng yếu giữa **UI Layer** và **Service Layer**, phản ánh tính phức tạp đặc thù của domain inventory. Thiết kế của controller này thể hiện việc áp dụng nhiều pattern và nguyên tắc kỹ thuật nâng cao:

1. **Quản lý Thực thể và Giao diện Đa chiều (Entity & Interface Management):** Controller thực hiện **Dual Entity Management**, quản lý song song hai thực thể `InventoryItem` và `InventoryTransaction` thông qua hai hệ thống **Observable Lists** và **TableViews** riêng biệt. Đồng thời, kiến trúc **Dual Form Management** được áp dụng để tách biệt ngữ cảnh xử lý: **Item form** chuyên biệt cho các **CRUD Operations** và **Stock form** dành riêng cho các **Stock in/out Operations**.
2. **Đồng bộ Dữ liệu và Xử lý Nghiệp vụ (Synchronization & Business Logic):** Tính nhất quán dữ liệu được đảm bảo qua cơ chế **Integrated Updates** và

Dynamic Combo Box Population: dữ liệu trong Item combo box (thuộc Stock form) được **Populate** động và tự động **Refresh** ngay khi danh sách items thay đổi. Logic nghiệp vụ được tập trung tại `handleStockOperation()`, thể hiện một **Complex Business Operation** với quy trình validation chặt chẽ, xử lý **Data Type Handling** kiểu `Double` (cho quantity/threshold) và điều phối gọi các **Service Methods** khác nhau dựa trên transaction type.

3. **Tối ưu hóa Truy vấn và Vấn đề Bảo trì (Query Optimization & Maintenance):** Về mặt hiển thị, controller ứng dụng **Advanced Filtering** thông qua `FilteredList` và **Predicate Function**, cho phép lọc dữ liệu đa tiêu chí (search, category, low stock). Tuy nhiên, về mặt cấu trúc mã nguồn, hệ thống vẫn tồn tại **Code Duplication** tại các phương thức hiển thị thông báo (`showSuccessAlert`, `showErrorAlert`), đặt ra yêu cầu cần **Refactor** để tối ưu hóa khả năng tái sử dụng.

3.3.4 Phân tích ShiftController.java

'ShiftController' là controller chịu trách nhiệm quản lý các tương tác liên quan đến ca làm việc (shift) trong hệ thống nhà hàng. Controller này xử lý các thao tác CRUD cho entity Shift và có dependency đặc biệt vào 'EmployeeService' để lấy danh sách nhân viên cho combo box. Điểm đặc biệt của controller này là hỗ trợ calendar view trong 'MainApp' thông qua method 'getShiftsByDate()', và có khả năng refresh employee combo box khi employees thay đổi.

3.3.4.1 Cấu trúc Class và khai báo Fields

a) Package và Import Statements

```
package com.restaurantmanagement.controller;

import com.restaurantmanagement.model.Employee;
import com.restaurantmanagement.model.Shift;
import com.restaurantmanagement.service.EmployeeService;
import com.restaurantmanagement.service.ShiftService;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;
```

```
import javafx.scene.control.DatePicker;
import javafx.scene.control.TableView;
import javafx.scene.control.TextField;

import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
import java.util.stream.Collectors;
```

Đoạn mã bắt đầu bằng việc khai báo package `com.restaurantmanagement.controller` và import các thành phần cốt lõi. Điểm đặc biệt của controller này là nó phụ thuộc vào cả hai service (`EmployeeService`, `ShiftService`) cùng các entity `Employee` và `Shift` để xử lý logic, thay vì chỉ một service như thông thường. Bên cạnh các thành phần giao diện JavaFX (như `DatePicker`, `ComboBox`, Collections), mã nguồn còn sử dụng `java.time` để xử lý thời gian và `stream.Collectors`, cho thấy việc áp dụng Stream API để thao tác dữ liệu danh sách theo phong cách functional programming.

b) Khai báo Class và Javadoc

```
/**
 * Controller for Shift management UI.
 * Handles user interactions and updates the view.
 */
public class ShiftController {
```

Tương tự cấu trúc chung của dự án, dòng 20-23 chứa JavaDoc mô tả mục đích sử dụng của lớp. Ngay sau đó, dòng 24 là lời khai báo `public class` chính thức để bắt đầu định nghĩa controller này.

c) Khai báo Field Service và Data

```
private final ShiftService shiftService;
private final EmployeeService employeeService;
private final ObservableList<Shift> shiftList;
private TableView<Shift> tableView;
```

Tại đoạn này, controller khai báo các dependency và thành phần dữ liệu cốt lõi. Ngoài `ShiftService` là service chủ đạo, controller đặc biệt tích hợp thêm

`EmployeeService` để truy xuất dữ liệu nhân viên (phục vụ các chức năng như populate combo box), thể hiện rõ sự tương tác chéo cần thiết giữa các domain dữ liệu. Cuối cùng, `ObservableList` và `TableView` được định nghĩa để quản lý và hiển thị danh sách các ca làm việc (`Shift`) lên giao diện người dùng.

d) Khai báo Field UI Components

```
// UI Components (for form inputs)
private TextField idField;
private ComboBox<String> employeeComboBox;
private DatePicker datePicker;
private ComboBox<Integer> startHourComboBox;
private ComboBox<Integer> startMinuteComboBox;
private ComboBox<Integer> endHourComboBox;
private ComboBox<Integer> endMinuteComboBox;
private ComboBox<String> shiftTypeComboBox;
```

Đoạn này tập trung khai báo các thành phần giao diện (UI) cho form nhập liệu. Cấu trúc form quản lý `Shift` phức tạp hơn đáng kể so với `EmployeeController`, đặc biệt ở cơ chế xử lý thời gian chi tiết. Ngoài các trường cơ bản như `idField` (chỉ đọc), chọn nhân viên và loại ca, giao diện sử dụng `DatePicker` kết hợp với tận **4 `ComboBox` riêng biệt** (giờ/phút cho thời gian bắt đầu và kết thúc) để đảm bảo kiểm soát chính xác dữ liệu nhập vào.

3.3.4.2 Constructor và khởi tạo

a) Constructor Không Tham số

```
public ShiftController() {
    this.shiftService = new ShiftService();
    this.employeeService = new EmployeeService();
    this.shiftList = FXCollections.observableArrayList();
    loadShifts();
}
```

Đoạn mã này định nghĩa **constructor public không tham số**, nơi thiết lập trạng thái ban đầu cho controller. Điểm đáng chú ý là controller **tự khởi tạo trực tiếp** (instantiate) cả `ShiftService` và `EmployeeService` thay vì nhận chúng từ bên ngoài (Dependency Injection), đồng nghĩa với việc các instance này hoạt động

độc lập và không chia sẻ dữ liệu với các controller khác. Cuối cùng, `shiftList` được khởi tạo và phương thức `loadShifts()` được gọi ngay lập tức để nạp dữ liệu ca làm việc vào giao diện khi khởi động.

b) Constructor với Dependency Injection

```
    /**
     * Constructor with shared EmployeeService instance.
     */
    public
ShiftController(com.restaurantmanagement.service.EmployeeService
employeeService) {
        this.shiftService = new ShiftService();
        this.employeeService = employeeService;
        this.shiftList = FXCollections.observableArrayList();
        loadShifts();
    }
```

Đoạn mã này giới thiệu một **constructor thứ hai (overloaded)** cho phép thực hiện **Dependency Injection** đối với `EmployeeService`.

Trong khi `ShiftService` vẫn được khởi tạo nội bộ (do không cần chia sẻ), việc inject `EmployeeService` từ bên ngoài là rất quan trọng để đảm bảo **tính nhất quán dữ liệu**: cả `ShiftController` và `EmployeeController` đều thao tác trên cùng một danh sách nhân viên. Các bước khởi tạo danh sách và tải dữ liệu (`loadShifts`) sau đó vẫn diễn ra tương tự như constructor mặc định.

3.3.4.3 Method setTableView(): Ràng buộc TableView

```
    /**
     * Set the TableView reference.
     */
    public void setTableView(TableView<Shift> tableView) {
        this.tableView = tableView;
        tableView.setItems(shiftList);
    }
```

Đoạn mã từ dòng 57 đến 61 thực hiện các thiết lập cơ bản cho `TableView` tương tự như các controller khác. Phương thức này lưu tham chiếu của bảng vào biến instance và thực hiện **data binding** bằng cách gán `shiftList` làm nguồn dữ liệu

(`setItems`), đảm bảo mọi thay đổi trong danh sách sẽ tự động cập nhật lên giao diện bảng.

3.3.4.4 Method setFormFields(): Đăng ký Form Fields và Populate Combo Boxes

```
    /**
     * Set form input fields.
     */
    public void setFormFields(TextField idField, ComboBox<String>
employeeComboBox,
                              DatePicker               datePicker,
ComboBox<Integer> startHourComboBox,
                              ComboBox<Integer>
startMinuteComboBox, ComboBox<Integer> endHourComboBox,
                              ComboBox<Integer> endMinuteComboBox,
ComboBox<String> shiftTypeComboBox) {
        this.idField = idField;
        this.employeeComboBox = employeeComboBox;
        this.datePicker = datePicker;
        this.startHourComboBox = startHourComboBox;
        this.startMinuteComboBox = startMinuteComboBox;
        this.endHourComboBox = endHourComboBox;
        this.endMinuteComboBox = endMinuteComboBox;
        this.shiftTypeComboBox = shiftTypeComboBox;

        // Populate employee combo box
        populateEmployeeComboBox();

        // Populate hour combo boxes (0-23)
        for (int i = 0; i < 24; i++) {
            startHourComboBox.getItems().add(i);
            endHourComboBox.getItems().add(i);
        }

        // Populate minute combo boxes (0, 15, 30, 45)
        for (int i = 0; i < 60; i += 15) {
            startMinuteComboBox.getItems().add(i);
            endMinuteComboBox.getItems().add(i);
        }

        // Set default values
        startHourComboBox.setValue(9);
        startMinuteComboBox.setValue(0);
        endHourComboBox.setValue(17);
        endMinuteComboBox.setValue(0);
```

```
        // Populate shift type combo box
        shiftTypeComboBox.getItems().addAll("Morning",
"Afternoon", "Evening", "Night");
    }
```

Phương thức `setFormFields` (dòng 65-99) thể hiện sự phức tạp trong quản lý form của `ShiftController` khi phải xử lý tới **9 tham số** đầu vào.

Ngoài việc lưu trữ tham chiếu các thành phần UI, phương thức này thực hiện logic khởi tạo dữ liệu rất chi tiết:

- **Nhân viên**: Ủy quyền cho phương thức riêng `populateEmployeeComboBox()` để lấy dữ liệu, giữ code gọn gàng.
- **Thời gian**: Sử dụng vòng lặp để điền tự động giờ (0-23) và phút (với bước nhảy 15 phút: 0, 15, 30, 45), đồng thời thiết lập mặc định chuẩn hành chính (9:00 17:00).
- **Loại ca**: Cung cấp sẵn 4 tùy chọn cố định (Morning, Afternoon, Evening, Night).

Cách thiết kế này đảm bảo tính hợp lý về nghiệp vụ (ví dụ: khung giờ 15 phút) và giúp người dùng thao tác nhanh hơn nhờ các giá trị mặc định.

3.3.4.5 Method populateEmployeeComboBox(): Populate Employee Combo Box

```
    /**
     * Populate employee combo box with available employees.
     */
    private void populateEmployeeComboBox() {
        if (employeeComboBox == null) {
            return;
        }
        List<Employee> employees = employeeService.getAllEmployees();
        employeeComboBox.getItems().clear();
        for (Employee emp : employees) {
            employeeComboBox.getItems().add(emp.getId()   +   "   "   +
emp.getName());
        }
    }
```

Đoạn mã từ dòng 106 đến 116 định nghĩa phương thức private `populateEmployeeComboBox`. Đây là một helper method quan trọng minh họa

cho mô hình **cross-domain data access**: controller sử dụng dependency phụ (`EmployeeService`) để truy xuất dữ liệu nhân viên thay vì service chính. Mã đảm bảo an toàn với việc kiểm tra `null` trước khi thực thi, sau đó làm sạch danh sách cũ và nạp lại dữ liệu nhân viên được định dạng chuỗi trực quan `"ID Name"` (ví dụ: "E001 John Doe"), giúp người dùng dễ dàng định danh chính xác nhân viên khi phân ca.

3.3.4.6 Method refreshEmployeeComboBox(): Public Method để Refresh Combo Box

```
    /**
     * Public method to refresh employee combo box.
     * Can be called when employees are added/updated/deleted.
     */
    public void refreshEmployeeComboBox() {
        populateEmployeeComboBox();
    }
```

Đoạn mã từ dòng 120 đến 123 định nghĩa phương thức **`refreshEmployeeComboBox()`** với phạm vi truy cập `public`, đóng vai trò như một cổng giao tiếp để các thành phần bên ngoài (cụ thể là `MainApp`) yêu cầu cập nhật lại danh sách nhân viên.

Về mặt kỹ thuật, phương thức này chỉ đơn giản là gọi lại hàm nội bộ `populateEmployeeComboBox()`, nhưng về mặt kiến trúc, nó giải quyết bài toán **đồng bộ dữ liệu giữa các module**. Khi có bất kỳ thay đổi nào từ phía `EmployeeController` (thêm, sửa, xóa nhân viên) hoặc khi người dùng chuyển tab, `MainApp` sẽ kích hoạt phương thức này. Điều này đảm bảo ComboBox chọn nhân viên trong form quản lý ca (`Shift`) luôn hiển thị dữ liệu mới nhất, thể hiện sự phối hợp chặt chẽ giữa các controller thông qua lớp điều phối trung gian.

3.3.4.7 Method loadShifts(): Tải Shifts từ Service

```
    /**
     * Load all shifts into the table.
     */
    public void loadShifts() {
        shiftList.clear();
```

```
        shiftList.addAll(shiftService.getAllShifts());
    }
```

Đoạn mã từ dòng 128 đến 132 định nghĩa phương thức `loadShifts()`, tuân theo quy trình chuẩn để làm mới dữ liệu hiển thị. Phương thức này thực hiện xóa sạch `shiftList` hiện tại và nạp lại toàn bộ dữ liệu mới nhất từ `ShiftService`. Cơ chế này đảm bảo `TableView` luôn đồng bộ với dữ liệu gốc ngay khi khởi tạo controller hoặc sau mỗi thao tác thêm, sửa, xóa (CRUD).

3.3.4.8 Method handleAdd(): Xử lý Thêm Shift

```
    /**
     * Handle add button click.
     */
    public void handleAdd() {
        try {
            Shift shift = createShiftFromForm();
            shiftService.addShift(shift);
            loadShifts();
            clearForm();
            populateEmployeeComboBox(); // Refresh employee list
            showSuccessAlert("Shift assigned successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error assigning shift", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 136 đến 147 xử lý chức năng `addShift` theo luồng CRUD tiêu chuẩn: bọc trong khối `try-catch`, chuyển đổi dữ liệu form thành đối tượng `Shift`, gọi service để lưu và cập nhật lại giao diện (`loadShifts`, `clearForm`). **Điểm khác biệt thú vị** là việc gọi lại `populateEmployeeComboBox()` ngay sau khi thêm ca. Mặc dù việc thêm `Shift` về mặt logic không làm thay đổi danh sách `Employee`, bước này mang tính chất "lập trình phòng thủ" (defensive programming), đảm bảo ComboBox luôn được đồng bộ hóa lại để đề phòng bất kỳ sự thay đổi ngầm định nào khác.

3.3.4.9 Method handleUpdate(): Xử lý Cập nhật Shift

```
    /**
     * Handle update button click.
     */
    public void handleUpdate() {
```

```
        try {
            Shift shift = createShiftFromForm();
            if (shift.getId() == null || shift.getId().isEmpty()) {
                showErrorAlert("Error", "Please select a shift to
update");
                return;
            }
            shiftService.updateShift(shift);
            loadShifts();
            clearForm();
            populateEmployeeComboBox(); // Refresh employee list
            showSuccessAlert("Shift updated successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error updating shift", e.getMessage());
        }
    }
```

Đoạn mã từ dòng 152 đến 167 cài đặt phương thức `updateShift`, về cơ bản kế thừa cấu trúc xử lý của `addShift` nhưng bổ sung bước **kiểm tra ID bắt buộc** (dòng 157-160). Bước validation này đảm bảo tính toàn vẹn dữ liệu, ngăn chặn việc cập nhật khi chưa chọn bản ghi cụ thể. Sau khi gọi `shiftService.updateShift()`, controller thực hiện chuỗi thao tác chuẩn hóa giao diện quen thuộc: làm mới danh sách bảng, xóa trắng form nhập liệu và nạp lại dữ liệu cho `employeeComboBox`, đảm bảo mọi thành phần UI đều phản ánh trạng thái mới nhất của hệ thống.

3.3.4.10 Method handleDelete(): Xử lý Xóa Shift

```
    /**
     * Handle delete button click.
     */
    public void handleDelete() {
        Shift selected =
tableView.getSelectionModel().getSelectedItem();
        if (selected == null) {
            showErrorAlert("Error", "Please select a shift to delete");
            return;
        }

        try {
            shiftService.deleteShift(selected.getId());
            loadShifts();
            clearForm();
            showSuccessAlert("Shift deleted successfully!");
        } catch (IllegalArgumentException e) {
            showErrorAlert("Error deleting shift", e.getMessage());
        }
```

```
    }
```

Quy trình xử lý tuân theo chuẩn tương tác UI: lấy đối tượng đang được chọn, kiểm tra hợp lệ, gọi `shiftService` để xóa theo ID, và cuối cùng là làm mới bảng và xóa form. Điểm tinh tế ở đây là phương thức này **không gọi lại `populateEmployeeComboBox()`**. Điều này cho thấy logic xử lý tối ưu hơn so với `add` hay `update`: việc xóa một ca làm việc hoàn toàn không tác động đến danh sách nhân viên, do đó việc nạp lại ComboBox là không cần thiết, giúp giảm bớt một thao tác xử lý thừa.

3.3.4.11 Method handleTableSelection(): Xử lý Lựa chọn Hàng trong Bảng

```
    /**
     * Handle table row selection.
     */
    public void handleTableSelection() {
        Shift selected =
tableView.getSelectionModel().getSelectedItem();
        if (selected != null) {
            populateForm(selected);
        }
    }
```


Đoạn mã từ dòng 192 đến 198 thực hiện logic **bind dữ liệu từ bảng ngược lại form** (Master-Detail view). Khi người dùng chọn một dòng trong `TableView`, phương thức này lấy đối tượng `Shift` đó và gọi `populateForm()` để hiển thị chi tiết. Đây là thao tác tiêu chuẩn để hỗ trợ việc xem chi tiết hoặc chuẩn bị cho thao tác cập nhật (Update), đảm bảo trải nghiệm người dùng mượt mà giống như các màn hình quản lý khác.

3.3.4.12 Method createShiftFromForm(): Tạo Shift Object từ Form

```
    /**
     * Create Shift object from form fields.
     */
    private Shift createShiftFromForm() {
        Shift shift = new Shift();
        shift.setId(idField.getText().trim());

        // Parse employee selection
        String employeeSelection = employeeComboBox.getValue();
```

```
            if (employeeSelection != null && !employeeSelection.isEmpty())
{
                String employeeId = employeeSelection.split(" ")[0];
                String employeeName = employeeSelection.contains(" ") ?
                    employeeSelection.substring(employeeSelection.indexOf("
") + 3) : "";

                // Get employee name from service
                Employee                        emp                         =
employeeService.getEmployeeById(employeeId).orElse(null);
                if (emp != null) {
                    shift.setEmployeeId(emp.getId());
                    shift.setEmployeeName(emp.getName());
                } else {
                    shift.setEmployeeId(employeeId);
                    shift.setEmployeeName(employeeName);
                }
            }

            shift.setDate(datePicker.getValue());

            // Create LocalTime from hour and minute combo boxes
            Integer startHour = startHourComboBox.getValue();
            Integer startMinute = startMinuteComboBox.getValue();
            if (startHour != null && startMinute != null) {
                shift.setStartTime(LocalTime.of(startHour, startMinute));
            }

            Integer endHour = endHourComboBox.getValue();
            Integer endMinute = endMinuteComboBox.getValue();
            if (endHour != null && endMinute != null) {
                shift.setEndTime(LocalTime.of(endHour, endMinute));
            }

            shift.setShiftType(shiftTypeComboBox.getValue());

            return shift;
        }
```

Đoạn mã từ dòng 202 đến 244 định nghĩa phương thức helper `createShiftFromForm()`, đóng vai trò cốt lõi trong việc chuyển đổi dữ liệu từ giao diện nhập liệu phức tạp thành đối tượng `Shift` hoàn chỉnh.

Điểm nhấn kỹ thuật nằm ở **logic xử lý nhân viên (Employee)**: thay vì chỉ lấy giá trị thô từ ComboBox, phương thức phân tích chuỗi định dạng `"ID Name"`, trích xuất ID và thực hiện tra cứu lại qua `EmployeeService`. Việc sử dụng `Optional` kết hợp logic fallback (dòng 215-224) đảm bảo tính toàn vẹn dữ liệu cao nhất: hệ

thống ưu tiên lấy đối tượng nhân viên thực tế từ database để đảm bảo tính nhất quán, nhưng vẫn giữ lại thông tin đã chọn trên giao diện nếu tra cứu thất bại.

Ngoài ra, việc xử lý thời gian thể hiện sự tỉ mỉ khi tổng hợp dữ liệu từ **4 ComboBox rời rạc** (giờ và phút cho thời gian bắt đầu/kết thúc) thành các đối tượng `LocalTime` chuẩn của Java. Cùng với việc gán ID, Ngày và Loại ca, phương thức này hoàn tất việc đóng gói dữ liệu đa chiều vào một entity duy nhất để sẵn sàng cho các thao tác CRUD.

3.3.4.13 Method createShiftFromForm(): Method getShiftsByDate(): Lấy Shifts theo Ngày

```
    /**
     * Get shifts for a specific date.
     */
    public List<Shift> getShiftsByDate(LocalDate date) {
        return shiftList.stream()
                .filter(shift -> shift.getDate() != null &&
shift.getDate().equals(date))
                .collect(Collectors.toList());
    }
```

Đoạn mã từ dòng 247 đến 253 định nghĩa phương thức `getShiftsByDate()`, một **query method** quan trọng đóng vai trò cung cấp dữ liệu tùy chỉnh cho các thành phần bên ngoài, cụ thể là để `MainApp` dựng giao diện lịch (Calendar View).

Thay vì trả về toàn bộ danh sách, phương thức này sử dụng sức mạnh của **Stream API** để thực hiện lọc dữ liệu (filtering) một cách tinh gọn và hiện đại. Bằng cách kết hợp giữa kiểm tra an toàn (`null check`) và so sánh điều kiện (`equals`), controller trích xuất chính xác các ca làm việc trùng khớp với ngày được chọn. Việc sử dụng `Collectors.toList()` ở bước cuối cùng không chỉ thể hiện phong cách lập trình hàm (functional programming) của Java 8+ mà còn giúp mã nguồn trở nên mạch lạc, dễ đọc hơn so với việc sử dụng các vòng lặp truyền thống.

Sự hiện diện của phương thức này cho thấy `ShiftController` không chỉ thực hiện các thao tác CRUD cơ bản mà còn đóng vai trò là một bộ điều phối dữ liệu

linh hoạt, hỗ trợ việc hiển thị thông tin đa dạng trên nhiều giao diện khác nhau trong ứng dụng.

3.3.4.14 Method populateForm(): Điền Form từ Shift Object

```
    /**
     * Populate form fields with selected shift data.
     */
    private void populateForm(Shift shift) {
        idField.setText(shift.getId());

        // Set employee in combo box
        String employeeDisplay = shift.getEmployeeId() + " " +
shift.getEmployeeName();
        employeeComboBox.setValue(employeeDisplay);

        datePicker.setValue(shift.getDate());

        // Set hour and minute combo boxes
        if (shift.getStartTime() != null) {
            startHourComboBox.setValue(shift.getStartTime().getHour());
            startMinuteComboBox.setValue(shift.getStartTime().getMinute
());
        }

        if (shift.getEndTime() != null) {
            endHourComboBox.setValue(shift.getEndTime().getHour());
            endMinuteComboBox.setValue(shift.getEndTime().getMinute());
        }

        shiftTypeComboBox.setValue(shift.getShiftType());
    }
```

Đoạn mã từ dòng 256 đến 279 hoàn tất chu trình dữ liệu bằng phương thức **`populateForm(Shift shift)`**. Đây là quá trình "chuyển đổi ngược" (reverse transformation), đưa dữ liệu từ đối tượng domain `Shift` hiển thị lên 9 thành phần giao diện nhập liệu.

Quy trình này được xử lý rất tỉ mỉ để đảm bảo tính đồng nhất:

- **Định dạng Nhân viên**: Tương tự như lúc lưu, dữ liệu nhân viên được tái cấu trúc thành chuỗi `"ID Name"` để khớp chính xác với các mục có sẵn trong `ComboBox`.
- **Phân tách Thời gian**: Đây là phần phức tạp nhất, mã nguồn thực hiện tách đối tượng `LocalTime` (giờ bắt đầu và kết thúc) thành các phần giờ và phút

riêng biệt bằng phương thức `.getHour()` và `.getMinute()`, sau đó mới gán vào 4 `ComboBox` tương ứng trên form.

– **Các trường còn lại**: ID, ngày (`DatePicker`) và loại ca làm việc được gán trực tiếp sau khi kiểm tra an toàn.

Cách tiếp cận này tương đồng với `BookingController` nhưng nâng cao hơn ở việc xử lý định dạng chuỗi tùy chỉnh cho nhân viên, đảm bảo rằng khi người dùng chọn một dòng trên bảng, toàn bộ thông tin phức tạp của ca làm việc sẽ xuất hiện chính xác và sẵn sàng để chỉnh sửa.

3.3.4.15 Method clearForm(): Xóa Form Fields

```
    /**
     * Clear form fields.
     */
    private void clearForm() {
        idField.clear();
        employeeComboBox.setValue(null);
        datePicker.setValue(LocalDate.now());
        startHourComboBox.setValue(9);
        startMinuteComboBox.setValue(0);
        endHourComboBox.setValue(17);
        endMinuteComboBox.setValue(0);
        shiftTypeComboBox.setValue(null);
        if (tableView != null) {
            tableView.getSelectionModel().clearSelection();
        }
    }
```

Đoạn mã từ dòng 282 đến 296 hoàn thiện trải nghiệm người dùng bằng phương thức **clearForm()**. Thay vì chỉ đơn thuần xóa trống các trường, phương thức này thực hiện việc đưa giao diện về một "trạng thái mặc định thông minh".

Cụ thể, các trường văn bản và lựa chọn trong bảng được xóa sạch, nhưng các thành phần liên quan đến thời gian được tái thiết lập về các giá trị chuẩn: DatePicker quay về ngày hiện tại (LocalDate.now()) và 4 ComboBox thời gian quay về khung giờ hành chính (09:00 17:00). Việc thiết lập các giá trị mặc định hợp lý này không chỉ giúp giao diện trông gọn gàng mà còn tối ưu hóa năng suất, giúp người dùng có

thể tạo một ca làm việc mới ngay lập tức mà không cần chỉnh sửa quá nhiều thông tin nếu đó là một ca làm việc tiêu chuẩn.

3.3.4.16 Method showSuccessAlert(): Hiển thị Thông báo Thành công

```
    /**
     * Show success alert.
     */
    private void showSuccessAlert(String message) {
        Alert alert = new Alert(Alert.AlertType.INFORMATION);
        alert.setTitle("Success");
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 299 đến 307 hoàn tất cấu trúc của controller bằng một phương thức chuẩn hóa, có đặc điểm và chức năng hoàn toàn tương đồng với các controller khác trong hệ thống. Việc duy trì tính nhất quán này giúp mã nguồn dễ bảo trì và mở rộng, đảm bảo rằng dù `ShiftController` có logic nghiệp vụ phức tạp về thời gian và nhân viên, nó vẫn tuân thủ đúng các quy tắc giao tiếp và vận hành chung của toàn bộ ứng dụng.

3.3.4.17 Method showErrorAlert(): Hiển thị Thông báo Lỗi

```
    /**
     * Show error alert.
     */
    private void showErrorAlert(String title, String message) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        alert.setTitle(title);
        alert.setHeaderText(null);
        alert.setContentText(message);
        alert.showAndWait();
    }
```

Đoạn mã từ dòng 310 đến 318 tiếp tục duy trì tính đồng bộ của dự án bằng cách triển khai phương thức cuối cùng với cấu trúc hoàn toàn tương tự như các controller khác. Việc giữ nguyên **Implementation** và **JavaDoc** chuẩn hóa ở phần cuối này đảm bảo rằng Controller tuân thủ đúng Interface hoặc Design Pattern chung của hệ thống (như Command Pattern hoặc Template Method). Điều này giúp các lập

trình viên khác trong đội ngũ dễ dàng nắm bắt và bảo trì mã nguồn, dù `ShiftController` có chứa những logic nghiệp vụ đặc thù về thời gian và nhân viên ở các phần trước.

3.3.4.18 Method getShiftList(): Getter cho Observable List

```
    /**
     * Get observable list for binding.
     */
    public ObservableList<Shift> getShiftList() {
        return shiftList;
    }
```

Đoạn mã từ dòng 321 đến 324 hoàn tất cấu trúc dữ liệu của controller bằng cách cung cấp phương thức getter để truy xuất **`shiftList`**. Tương tự như các controller khác trong hệ thống, việc trả về tham chiếu đến `ObservableList` cho phép các thành phần bên ngoài (như `MainApp` hoặc các lớp View) có thể quan sát và phản ứng trực tiếp với những thay đổi của dữ liệu ca làm việc mà không cần phải truy vấn lại service.

3.3.4.19 Tổng kết

`ShiftController` đóng vai trò là lớp điều khiển trung gian (Controller) trong mô hình kiến trúc MVC, chịu trách nhiệm điều phối luồng dữ liệu giữa giao diện người dùng (UI) và các dịch vụ nghiệp vụ (Services). Thực thể này thể hiện các nguyên lý thiết kế phần mềm và các mẫu định hình (Design Patterns) chuyên sâu sau:

1. **Cơ chế Quản lý Phụ thuộc (Dependency Management)**

– **Dual Service Dependency:** Khác với các controller đơn miền, `ShiftController` duy trì phụ thuộc vào cả `ShiftService` và `EmployeeService`. Việc tích hợp này cho phép thực hiện các truy vấn liên miền (cross-domain queries), đảm bảo ràng buộc dữ liệu giữa thực thể "Ca làm việc" và "Nhân viên".

– **Dual Constructor Pattern & Dependency Injection (DI):** Controller hỗ trợ cơ chế khởi tạo linh hoạt. Việc cung cấp Constructor cho phép Inject

`EmployeeService` từ bên ngoài giúp thực hiện chia sẻ trạng thái (shared state) với `EmployeeController`, đảm bảo tính nhất quán của dữ liệu (Data Consistency) trên toàn ứng dụng.

2. **Tương tác liên miền và đồng bộ dữ liệu**

– **Cross-Domain Data Access:** Thông qua `EmployeeService`, Controller thực hiện nạp dữ liệu động (Dynamic Population) cho các thành phần UI. Đây là minh chứng cho mối quan hệ phụ thuộc mức logic (Logical Dependency) giữa miền quản lý nhân sự và miền điều phối ca trực.

– **Event-driven Synchronization:** Phương thức `refreshEmployeeComboBox()` đóng vai trò là một Interface công khai, cho phép lớp điều phối ứng dụng (`MainApp`) kích hoạt việc cập nhật lại trạng thái giao diện khi có các sự kiện thay đổi dữ liệu từ các domain khác (thêm/xóa/sửa nhân viên).

3. **Xử lý Logic nghiệp vụ tại tầng Controller**

– **Complex Temporal Handling:** Việc phân rã và hợp nhất thời gian thông qua 4 trường dữ liệu thành phần (giờ/phút bắt đầu và kết thúc) đòi hỏi logic kiểm soát kiểu dữ liệu `LocalTime` chặt chẽ, nhằm đảm bảo tính toàn vẹn của dữ liệu thời gian trước khi chuyển xuống tầng Service.

– **Data Parsing & Transformation:** Controller thực hiện cơ chế phân tách chuỗi (Parsing) từ định dạng hiển thị ("ID Name") ngược trở lại đối tượng Business Object. Logic này bao gồm cả cơ chế dự phòng (fallback logic) để xử lý ngoại lệ khi dữ liệu gốc bị thay đổi trong quá trình tương tác.

4. **Tối ưu hóa truy vấn và hiển thị**

– **Functional Programming với Stream API:** Phương thức `getShiftsByDate()` tận dụng sức mạnh của Java Stream API để thực hiện lọc dữ liệu (Filtering) ngay tại bộ nhớ (In-memory), hỗ trợ cung cấp dữ liệu tức thời cho các thành phần giao diện phức tạp như Calendar View.

5. **Đánh giá và hướng cải thiện (Refactoring)**

- **Tính đóng gói (Encapsulation):** Controller đã thực hiện tốt việc tách biệt logic hiển thị và logic nghiệp vụ, tuy nhiên vẫn tồn tại sự lặp lại mã nguồn (Code Duplication) ở các phương thức thông báo (Alerts).
- **Kiến nghị:** Áp dụng **Inheritance** (Kế thừa) bằng cách xây dựng một `BaseController` để tái sử dụng các phương thức dùng chung.
- Chuyển đổi các giá trị hằng số (Hardcoded values) của loại ca làm việc sang dạng **Enumeration (Enum)** để tăng tính an toàn về kiểu (Type-safety).

Kết luận: `ShiftController` không chỉ đơn thuần là bộ xử lý sự kiện giao diện mà còn đóng vai trò là một **Integrator** (Bộ tích hợp) quan trọng, đảm bảo trải nghiệm người dùng (UX) liền mạch và tính nhất quán dữ liệu giữa các module độc lập trong hệ thống.

3.4 Tầng Service

Tầng Service, được đại diện bởi các class service trong biểu đồ UML, đóng vai trò là lớp xử lý business logic và điều phối truy cập dữ liệu thông qua repository. Trong kiến trúc phân lớp được định nghĩa ở Chương 2, tầng này nằm giữa Tầng Controller và Tầng Repository, chịu trách nhiệm thực hiện các quy tắc nghiệp vụ, validation dữ liệu, và quản lý các thao tác phức tạp trên entities. Phân tích chi tiết này xem xét từng thành phần của source code các service với các đoạn code cụ thể và giải thích từng dòng để chứng minh cách thiết kế kiến trúc được chuyển đổi thành hiện thực thực tế.

3.4.1 Phân tích BookingService.java

'BookingService' là service chịu trách nhiệm xử lý business logic cho domain Booking (đặt bàn) trong hệ thống nhà hàng. Service này thực hiện các thao tác CRUD cơ bản, validation dữ liệu, và các thao tác nghiệp vụ đặc biệt như hủy đặt bàn và xác nhận khách đã đến. Service này đóng vai trò là lớp trung gian giữa 'BookingController' và 'InMemoryBookingRepository', đảm bảo business rules được thực thi trước khi dữ liệu được lưu trữ.

3.4.1.1 Cấu trúc Class và Khai báo Field

a) Package và Import Statements

```
import com.restaurantmanagement.model.Booking;
import
com.restaurantmanagement.repository.InMemoryBookingRepository;
import java.time.LocalDate;
import java.util.List;
import java.util.Optional;
```

Đoạn mã xác định package ‘com.restaurantmanagement.service’, phù hợp với kiến trúc phân lớp trong đó service thuộc tầng xử lý business logic. Lớp ‘Booking’ được import từ domain model, trong khi ‘InMemoryBookingRepository’ được sử dụng để truy cập dữ liệu từ tầng repository, thể hiện sự phụ thuộc trực tiếp vào một implementation cụ thể. Ngoài ra, các lớp tiện ích chuẩn của Java như ‘LocalDate’, ‘List’ và ‘Optional’ được sử dụng để xử lý ngày tháng, danh sách và các giá trị có thể không tồn tại.

b) Khai báo Class và JavaDoc

```
/
  Service layer for Booking business logic.
 /
public class BookingService
```

JavaDoc tại các dòng 9–11 mô tả chức năng của lớp service, nhấn mạnh vai trò xử lý nghiệp vụ (business logic) liên quan đến Booking, thay vì chỉ đơn thuần thực hiện truy cập dữ liệu. Tại dòng 12, class được khai báo với phạm vi ‘public’, cho phép các lớp ở package khác, đặc biệt là ‘BookingController’ trong package ‘controller’, có thể truy cập và sử dụng.

c) Khai báo Field Repository

```
private final InMemoryBookingRepository repository;
```

Tại dòng 13, thuộc tính ‘private final InMemoryBookingRepository repository’ được khai báo nhằm thể hiện dependency của service vào tầng repository. Từ khóa ‘private’ giới hạn phạm vi truy cập trong nội bộ class, trong khi ‘final’ đảm bảo dependency không bị thay đổi sau khi khởi tạo, góp phần duy trì tính bất biến.

Kiểu ‘InMemoryBookingRepository’ cho thấy service đang sử dụng một implementation cụ thể để truy cập dữ liệu booking.

Tuy nhiên, việc phụ thuộc trực tiếp vào implementation thay vì interface làm giảm tính linh hoạt. Giải pháp phù hợp hơn là sử dụng interface ‘BookingRepository’, cho phép dễ dàng thay đổi cơ chế lưu trữ (ví dụ từ in-memory sang database) mà không cần chỉnh sửa mã nguồn của service.

3.4.1.2 Constructor và khởi tạo

```
public BookingService() {
 this.repository = new InMemoryBookingRepository();
}
```

Tại dòng 15, constructor ‘public’ không tham số được khai báo nhằm cho phép controller dễ dàng khởi tạo instance của service. Dòng 16 thực hiện khởi tạo repository thông qua câu lệnh ‘this.repository = new InMemoryBookingRepository()’, trong đó ‘this.repository’ tham chiếu đến thuộc tính của class và một instance mới của ‘InMemoryBookingRepository’ được tạo ra và gán cho thuộc tính này.

Cách tiếp cận này cho thấy service tự khởi tạo repository thay vì nhận thông qua dependency injection. Mặc dù giúp đơn giản hóa việc sử dụng trong kiến trúc hiện tại, cách làm này làm giảm tính linh hoạt và khả năng kiểm thử. Trong các kiến trúc phức tạp hơn, repository thường được inject qua constructor để dễ dàng thay đổi implementation và hỗ trợ testing hiệu quả hơn.

3.4.1.3 Method addBooking(): Thêm Booking Mới

```
/
  Add a new booking.
 /
public Booking addBooking(Booking booking) {
 validateBooking(booking);
 return repository.save(booking);
}
```

Các dòng 19–21 sử dụng JavaDoc để mô tả rõ mục đích và cách sử dụng của phương thức ‘addBooking’. Phương thức này được thiết kế với phạm vi truy cập ‘public’, cho phép được gọi từ ‘BookingController.handleAdd()’, nhận vào một đối tượng ‘Booking’ và trả về đối tượng ‘Booking’ sau khi đã được lưu trữ. Tại dòng 22, phương thức ‘addBooking(Booking booking)’ được khai báo với kiểu trả về là ‘Booking’, thể hiện kết quả của thao tác thêm mới booking vào hệ thống.

Dòng 23 thực hiện gọi phương thức ‘validateBooking(booking)’ nhằm kiểm tra tính hợp lệ của dữ liệu đầu vào trước khi tiến hành lưu trữ. Cơ chế này áp dụng nguyên tắc fail-fast, trong đó các dữ liệu không hợp lệ sẽ được phát hiện sớm thông qua việc ném ra ‘IllegalArgumentException’. Dòng 24 ủy quyền thao tác lưu dữ liệu cho tầng repository thông qua phương thức ‘repository.save(booking)’ và trả về kết quả tương ứng. Repository chịu trách nhiệm xử lý chi tiết lưu trữ, bao gồm cả việc tự động sinh ID nếu cần.

Nhìn chung, phương thức này thể hiện rõ vai trò của tầng service trong kiến trúc hệ thống: đảm bảo dữ liệu hợp lệ và điều phối luồng xử lý, trong khi logic lưu trữ được giao cho repository, phù hợp với mô hình thin service.

3.4.1.4 Method updateBooking(): Cập nhật Booking

```
     /
       Update an existing booking.
      /
     public Booking updateBooking(Booking booking) {
      if (booking.getId() == null || booking.getId().isEmpty()) {
       throw new IllegalArgumentException("Booking ID is required
for update");
      }
      if (!repository.existsById(booking.getId())) {
       throw new IllegalArgumentException("Booking with ID " +
booking.getId() + " not found");
      }
      validateBooking(booking);
```

```
 return repository.save(booking);
}
```

Các dòng 27–29 khai báo một phương thức ‘public’ được gọi từ ‘BookingController.handleUpdate()’, nhận vào đối tượng ‘Booking’ đã có ID để thực hiện cập nhật. Tại các dòng 30–32, phương thức kiểm tra tính hợp lệ của ID, đảm bảo ID không null hoặc rỗng. Nếu điều kiện không thỏa mãn, ‘IllegalArgumentException’ được ném ra với thông báo rõ ràng, nhằm đảm bảo thao tác cập nhật chỉ áp dụng cho booking đã tồn tại.

Các dòng 33–35 tiếp tục xác nhận sự tồn tại của booking trong hệ thống thông qua ‘repository.existsById()’. Trường hợp booking không tồn tại, phương thức sẽ ném exception kèm thông tin ID, ngăn chặn việc tạo mới thông qua thao tác update. Dòng 36 thực hiện kiểm tra tính hợp lệ của dữ liệu booking bằng cách gọi ‘validateBooking(booking)’.

Cuối cùng, tại dòng 37, phương thức gọi ‘repository.save(booking)’ để cập nhật dữ liệu và trả về đối tượng booking sau khi hoàn tất. Tổng thể, phương thức áp dụng nguyên tắc defensive programming với nhiều bước kiểm tra nhằm đảm bảo tính an toàn và nhất quán của dữ liệu trước khi thực hiện cập nhật.

3.4.1.5 Method deleteBooking(): Xóa Booking

```
/
  Delete a booking by ID.
 /
public boolean deleteBooking(String id) {
    if (id == null || id.isEmpty()) {
        throw new IllegalArgumentException("Booking ID cannot be empty");
    }
    return repository.deleteById(id);
}
```

Phương thức xóa booking có phạm vi truy cập ‘public’, nhận tham số ‘String id’ và trả về ‘boolean’, trong đó ‘true’ khi xóa thành công và ‘false’ khi không tìm thấy booking tương ứng. Phương thức được gọi từ

‘BookingController.handleCancel()’ hoặc một phương thức xóa riêng. Trước khi xóa, ID được kiểm tra không ‘null’ hoặc rỗng; nếu không hợp lệ, phương thức ném exception với thông báo rõ ràng. Sau đó, thao tác xóa được thực hiện thông qua ‘repository.deleteById(id)’, và kết quả xóa được trả về. So với ‘updateBooking()’, phương thức này đơn giản hơn do chỉ cần validate ID, không cần kiểm tra toàn bộ đối tượng booking. Đây là pattern simple operation cho thao tác xóa dựa trên ID.

3.4.1.6 Method getBookingById(): Lấy Booking theo ID

```
/
  Get booking by ID.
 /
public Optional<Booking> getBookingById(String id) {
    return repository.findById(id);
}
```

Phương thức có phạm vi truy cập ‘public’, dùng để truy vấn thông tin booking theo ID và trả về ‘Optional<Booking>‘. Kết quả trả về chứa đối tượng ‘Booking’ nếu tìm thấy, hoặc ‘Optional.empty()’ nếu không tồn tại, giúp tránh việc trả về ‘null’ và tăng tính an toàn khi xử lý dữ liệu. Phương thức gọi trực tiếp ‘repository.findById(id)’ và trả về kết quả mà không bổ sung validation hay business logic. Đây là pattern pass-through, trong đó service chỉ đóng vai trò trung gian chuyển tiếp yêu cầu đến repository. Việc sử dụng ‘Optional’ là best practice trong Java 8+ nhằm hạn chế lỗi NullPointerException.

3.4.1.7 Method getAllBookings(): Lấy Tất cả Bookings

```
/
  Get all bookings.
 /
public List<Booking> getAllBookings() {
    return repository.findAll();
}
```

Phương thức có phạm vi truy cập ‘public’, được gọi từ ‘BookingController.loadBookings()’, dùng để lấy toàn bộ danh sách booking trong

hệ thống và trả về ‘List<Booking>‘. Phương thức gọi trực tiếp ‘repository.findAll()’ và trả về kết quả mà không bổ sung logic xử lý như lọc hay sắp xếp. Đây là pattern pass-through, trong đó service chỉ đóng vai trò chuyển tiếp yêu cầu đến repository, phục vụ việc hiển thị danh sách booking trong bảng tại controller.

3.4.1.8 Method getBookingsByDate(): Lấy Bookings theo Ngày

```
    /
      Get bookings by date.
     /
    public List<Booking> getBookingsByDate(LocalDate date) {
        return repository.findByDate(date);
    }
```

Phương thức có phạm vi truy cập ‘public’, dùng để truy vấn các booking theo ngày, nhận tham số ‘LocalDate date’ và trả về ‘List<Booking>‘ tương ứng. Phương thức gọi ‘repository.findByDate(date)’ để lấy danh sách các booking có ngày trùng với tham số. Đây là một query method không chứa business logic phức tạp; service chỉ đóng vai trò abstraction, giúp controller không phụ thuộc trực tiếp vào cơ chế lọc dữ liệu của repository.

3.4.1.9 Method getBookingsByStatus(): Lấy Bookings theo Trạng thái

```
    /
      Get bookings by status.
     /
    public List<Booking> getBookingsByStatus(String status) {
        return repository.findByStatus(status);
    }
```

Phương thức có phạm vi truy cập ‘public’, dùng để truy vấn booking theo trạng thái, nhận tham số ‘String status’ (CONFIRMED, SEATED, CANCELLED) và trả về ‘List<Booking>‘. Phương thức gọi ‘repository.findByStatus(status)’ để lọc và trả về danh sách các booking có trạng thái phù hợp. Đây là một query method tương tự ‘getBookingsByDate()’, cho phép controller lấy và hiển thị các booking theo từng trạng thái cụ thể..

3.4.1.10 Method searchBookingsByName(): Tìm kiếm Bookings theo Tên Khách hàng

```
    /
      Search bookings by customer name.
     /
    public List<Booking> searchBookingsByName(String name) {
        return repository.findByCustomerName(name);
    }
```

Phương thức có phạm vi truy cập 'public', dùng để tìm kiếm booking theo tên khách hàng, nhận tham số 'String name' và trả về 'List<Booking>'. Phương thức gọi 'repository.findByCustomerName(name)' để lọc các booking có tên khách hàng khớp (chính xác hoặc một phần, tùy implementation). Đây là một query method cung cấp chức năng tìm kiếm cho controller và có thể được sử dụng trong tương lai để mở rộng tính năng search.

3.4.1.11 Method cancelBooking(): Hủy Booking

```
    /
      Cancel a booking.
     /
    public Booking cancelBooking(String id) {
        Optional<Booking> bookingOpt = repository.findById(id);
        if (!bookingOpt.isPresent()) {
            throw new IllegalArgumentException("Booking with ID "
+ id + " not found");
        }

        Booking booking = bookingOpt.get();
        if (!booking.canCancel()) {
            throw new IllegalArgumentException("Booking cannot be
cancelled. Current status: " + booking.getStatus());
        }
```

```
        booking.setStatus("CANCELLED");
        return repository.save(booking);
    }
```

Phương thức có phạm vi truy cập ‘public’, dùng để xử lý nghiệp vụ hủy booking, nhận tham số ‘String id’ và trả về đối tượng ‘Booking’ đã được cập nhật. Đây là một business operation, không phải thao tác CRUD đơn giản.

Phương thức truy vấn booking theo ID thông qua ‘repository.findById(id)’ và sử dụng ‘Optional’ để xử lý an toàn trường hợp không tồn tại. Nếu không tìm thấy booking, phương thức ném exception kèm theo ID. Khi booking tồn tại, phương thức kiểm tra business rule thông qua ‘booking.canCancel()’ để đảm bảo booking ở trạng thái hợp lệ cho phép hủy; nếu không thỏa mãn, exception sẽ được ném kèm trạng thái hiện tại.

Sau khi vượt qua các kiểm tra, trạng thái booking được cập nhật thành ‘"CANCELLED"‘ và thay đổi được lưu lại bằng ‘repository.save(booking)’. Phương thức thể hiện rõ business operation pattern, bao gồm kiểm tra tồn tại, kiểm tra nghiệp vụ, cập nhật trạng thái và lưu dữ liệu.

3.4.1.12 Method seatCustomer(): Xác nhận Khách Đã Đến

```
    /
      Seat a customer (change status from CONFIRMED to SEATED).
     /
    public Booking seatCustomer(String id) {
        Optional<Booking> bookingOpt = repository.findById(id);
        if (!bookingOpt.isPresent()) {
            throw new IllegalArgumentException("Booking with ID "
+ id + " not found");
        }

        Booking booking = bookingOpt.get();
        if (!booking.canSeat()) {
```

```
            throw new IllegalArgumentException("Booking cannot be
seated. Current status: " + booking.getStatus());
        }

        booking.setStatus("SEATED");
        return repository.save(booking);
    }
```

Phương thức có phạm vi truy cập ‘public’, dùng để xử lý nghiệp vụ xác nhận khách đã đến, nhận tham số ‘String id’ và trả về đối tượng ‘Booking’ đã được cập nhật. JavaDoc mô tả rõ việc chuyển trạng thái từ ‘CONFIRMED’ sang ‘SEATED’.

Phương thức truy vấn booking theo ID thông qua ‘Optional’, kiểm tra sự tồn tại tương tự ‘cancelBooking()’. Sau đó, business rule được kiểm tra bằng ‘booking.canSeat()’ để đảm bảo booking ở trạng thái hợp lệ; nếu không, phương thức ném exception kèm trạng thái hiện tại. Khi các điều kiện được thỏa mãn, trạng thái booking được cập nhật thành ‘"SEATED"‘ và thay đổi được lưu lại bằng repository. Phương thức có cấu trúc tương tự ‘cancelBooking()’, khác biệt ở business rule áp dụng và trạng thái mới, thể hiện pattern similar business operations.

3.4.1.13 Method validateBooking(): Validation Dữ liệu

```
    /
      Validate booking data.
     /
    private void validateBooking(Booking booking) {
        if (booking == null) {
            throw new IllegalArgumentException("Booking cannot be null");
        }
        if (booking.getCustomerName() == null ||
booking.getCustomerName().trim().isEmpty()) {
            throw new IllegalArgumentException("Customer name is required");
        }
```

```
            if (booking.getPhoneNumber() == null ||
booking.getPhoneNumber().trim().isEmpty()) {
                throw new IllegalArgumentException("Phone number is required");
            }
            if (booking.getNumberOfGuests() <= 0) {
                throw new IllegalArgumentException("Number of guests must be
greater than 0");
            }
            if (booking.getDate() == null) {
                throw new IllegalArgumentException("Date is required");
            }
            if (booking.getStartTime() == null) {
                throw new IllegalArgumentException("Time is required");
            }
            if (booking.getTableId() == null ||
booking.getTableId().trim().isEmpty()) {
                throw new IllegalArgumentException("Table ID is required");
            }
            if (booking.getStatus() == null ||
booking.getStatus().trim().isEmpty()) {
                throw new IllegalArgumentException("Status is required");
            }
        }
```

Phương thức private dùng để validate dữ liệu booking, được gọi từ addBooking() và updateBooking(). Việc tách validation thành phương thức riêng giúp tránh lặp code và tăng khả năng tái sử dụng. Phương thức kiểm tra đối tượng booking không null, sau đó thực hiện validation toàn diện các trường dữ liệu như tên khách hàng, số điện thoại, số lượng khách, ngày, giờ bắt đầu, mã bàn và trạng thái. Các trường chuỗi được kiểm tra không null, không rỗng và không chỉ chứa khoảng trắng; các trường số và thời gian được kiểm tra tính hợp lệ tương ứng. Cách tiếp cận này thể hiện comprehensive validation pattern, đảm bảo dữ liệu hợp lệ trước khi lưu

và đặt validation tại service layer nhằm áp dụng nhất quán cho mọi controller, đồng thời tách biệt logic nghiệp vụ khỏi tầng giao diện.

3.4.1.14 Tổng kết BookingService

BookingService thể hiện rõ các pattern và nguyên tắc thiết kế quan trọng trong kiến trúc ứng dụng.

Trước hết, service áp dụng Repository Pattern để truy cập dữ liệu thông qua repository, qua đó tách biệt rõ ràng giữa business logic và data access logic. Bên cạnh đó, service đóng vai trò validation layer, thực hiện kiểm tra dữ liệu đầu vào trước khi lưu nhằm đảm bảo tính toàn vẹn dữ liệu và tuân thủ các business rules.

Ngoài các thao tác CRUD cơ bản, BookingService còn cung cấp các business operations như cancelBooking() và seatCustomer(). Đây là những nghiệp vụ có logic xử lý phức tạp, bao gồm kiểm tra trạng thái, áp dụng rule nghiệp vụ và cập nhật dữ liệu. Một phần business logic được ủy quyền cho model thông qua các phương thức như booking.canCancel() và booking.canSeat(), thể hiện cách tiếp cận rich domain model.

Service cũng cung cấp nhiều query methods để hỗ trợ các use case khác nhau, chẳng hạn truy vấn booking theo ngày, theo trạng thái hoặc theo tên khách hàng. Việc sử dụng Optional giúp xử lý an toàn các trường hợp không tìm thấy entity, hạn chế lỗi NullPointerException. Đồng thời, các lỗi validation và vi phạm nghiệp vụ được thông báo thông qua exception handling với thông điệp rõ ràng.

Một số phương thức đơn giản được triển khai theo thin service (pass-through) pattern, chỉ chuyển tiếp yêu cầu đến repository mà không bổ sung logic xử lý. Nhìn chung, BookingService áp dụng defensive programming với nhiều bước kiểm tra cần thiết trước khi thực hiện thao tác.

BookingService đóng vai trò là lớp trung gian bảo vệ dữ liệu và business rules, đảm bảo chỉ các thao tác hợp lệ mới được thực hiện, đồng thời cung cấp abstraction cho controller để controller không phụ thuộc vào chi tiết triển khai của repository.

3.4.2 Phân tích EmployeeService.java

'EmployeeService' là service chịu trách nhiệm xử lý business logic cho domain Employee (nhân viên) trong hệ thống nhà hàng. Service này thực hiện các thao tác CRUD cơ bản và validation dữ liệu cho entity Employee. So với 'BookingService', service này đơn giản hơn vì không có các business operations đặc biệt như cancel hay seat customer, chỉ tập trung vào quản lý thông tin nhân viên cơ bản.

3.4.2.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.service;


import com.restaurantmanagement.model.Employee;
import
com.restaurantmanagement.repository.InMemoryEmployeeRepository;
import java.util.List;
import java.util.Optional;
```

Các dòng đầu của 'EmployeeService' khai báo package 'com.restaurantmanagement.service', nhất quán với cấu trúc của 'BookingService'. Lớp 'Employee' được import từ package model, đại diện cho entity nhân viên trong domain model. Service sử dụng 'InMemoryEmployeeRepository' từ package repository để truy cập dữ liệu, tương tự cách 'BookingService' làm việc với booking repository. Ngoài ra, 'List' và 'Optional' từ 'java.util' được import để hỗ trợ trả về danh sách nhân viên và xử lý an toàn trường hợp không tìm thấy employee. So với 'BookingService', 'EmployeeService' có số lượng import ít hơn do không cần xử lý các kiểu dữ liệu như 'LocalDate', vì không cung cấp các query methods theo ngày.

b) Khai báo Class và JavaDoc

```
/
  Service layer for Employee business logic.
 /
```

```
public class EmployeeService
```

JavaDoc tại các dòng 8–10 mô tả ngắn gọn vai trò của ‘EmployeeService’ là service layer chịu trách nhiệm xử lý business logic liên quan đến employee, tương tự cách ‘BookingService’ được thiết kế. Tại dòng 11, class được khai báo với phạm vi truy cập ‘public’, cho phép các lớp ở package khác, đặc biệt là controller, có thể sử dụng service này.

c) Khai báo Field Repository

```
private final InMemoryEmployeeRepository repository;
```

Tại dòng 12, thuộc tính ‘private final InMemoryEmployeeRepository repository’ được khai báo để thể hiện sự phụ thuộc của service vào repository. Cách tiếp cận này tương tự ‘BookingService’, trong đó service làm việc trực tiếp với một implementation cụ thể của repository. Từ khóa ‘final’ đảm bảo dependency không bị thay đổi sau khi khởi tạo, góp phần tăng tính an toàn và nhất quán trong thiết kế.

3.4.2.2 Constructor và khởi tạo

```
public EmployeeService() {
    this.repository = new InMemoryEmployeeRepository();
}
```

Constructor tại dòng 14 được khai báo với phạm vi truy cập ‘public’ và không nhận tham số, tương tự ‘BookingService’. Tại dòng 15, service khởi tạo trực tiếp ‘InMemoryEmployeeRepository’ thông qua ‘new’, cho thấy service tự chịu trách nhiệm tạo repository instance. Cách tiếp cận này đơn giản và dễ triển khai, nhưng làm giảm tính linh hoạt so với các giải pháp như dependency injection. Việc constructor của ‘EmployeeService’ được thiết kế giống hoàn toàn ‘BookingService’ thể hiện sự nhất quán trong kiến trúc các service.

3.4.2.3 Method addEmployee(): Thêm Employee Mới

```
/
  Add a new employee.
 /
public Employee addEmployee(Employee employee) {
```

```
        validateEmployee(employee);
        return repository.save(employee);
    }
```

JavaDoc tại các dòng 18–20 mô tả method ‘public’ được gọi từ ‘EmployeeController.handleAdd()’, nhận vào đối tượng ‘Employee’ và trả về ‘Employee’ đã được lưu. Tại dòng 21, phương thức ‘validateEmployee(employee)’ được gọi trước khi lưu nhằm đảm bảo dữ liệu hợp lệ, áp dụng pattern fail-fast tương tự ‘BookingService’. Dòng 22 gọi ‘repository.save(employee)’ để thực hiện lưu dữ liệu; repository chịu trách nhiệm tạo ID nếu employee chưa có và trả về đối tượng đã được lưu. Phương thức này có cấu trúc hoàn toàn tương tự ‘BookingService.addBooking()’, chỉ khác về entity và phương thức validation, thể hiện pattern consistent design trong các service.

3.4.2.4 Method updateEmployee(): Cập nhật Employee

```
    /
      Update an existing employee.
     /
    public Employee updateEmployee(Employee employee) {
        if (employee.getId() == null || employee.getId().isEmpty()) {
            throw new IllegalArgumentException("Employee ID is
required for update");
        }
        if (!repository.existsById(employee.getId())) {
            throw new IllegalArgumentException("Employee with ID " +
employee.getId() + " not found");
        }
        validateEmployee(employee);
        return repository.save(employee);
    }
```

JavaDoc tại các dòng 26–28 mô tả phương thức ‘public’ được gọi từ ‘EmployeeController.handleUpdate()’, nhận vào đối tượng ‘Employee’ đã có ID để

thực hiện cập nhật. Phương thức trước hết kiểm tra tính hợp lệ của ID, đảm bảo ID không ‘null’ hoặc rỗng; nếu không hợp lệ sẽ ném exception. Tiếp theo, service kiểm tra sự tồn tại của employee thông qua ‘repository.existsById(employee.getId())’; nếu không tìm thấy, exception được ném kèm theo ID. Sau đó, dữ liệu employee được validate bằng ‘validateEmployee(employee)’ và cuối cùng được lưu lại thông qua ‘repository.save(employee)’. Phương thức này có cấu trúc và logic tương tự ‘BookingService.updateBooking()’, chỉ khác về entity, thể hiện sự nhất quán trong thiết kế khi áp dụng cùng một pattern cho các operations tương tự.

3.4.2.5 Method deleteEmployee(): Xóa Employee

```
     /
       Delete an employee by ID.
      /
     public boolean deleteEmployee(String id) {
         if (id == null || id.isEmpty()) {
             throw new IllegalArgumentException("Employee ID
cannot be empty");
         }
         return repository.deleteById(id);
     }
```

JavaDoc tại các dòng 40–42 mô tả phương thức ‘public’ được gọi từ ‘EmployeeController.handleDelete()’, nhận tham số ‘String id’ và trả về ‘boolean’, trong đó ‘true’ khi xóa thành công và ‘false’ khi không tìm thấy employee tương ứng. Phương thức thực hiện kiểm tra tính hợp lệ của ID, đảm bảo ID không ‘null’ hoặc rỗng; nếu không hợp lệ sẽ ném exception. Sau đó, thao tác xóa được thực hiện thông qua ‘repository.deleteById(id)’ và trả về kết quả. Phương thức này có cấu trúc hoàn toàn tương tự ‘BookingService.deleteBooking()’, thể hiện pattern thiết kế nhất quán giữa các service.

3.4.2.6 Method getEmployeeById(): Lấy Employee theo ID

```
     /
```

```
     Get employee by ID.
    /
    public Optional<Employee> getEmployeeById(String id) {
        return repository.findById(id);
    }
```

JavaDoc tại các dòng 50–52 mô tả phương thức ‘public’ dùng để truy vấn employee theo ID và trả về ‘Optional<Employee>‘ nhằm xử lý an toàn trường hợp không tìm thấy. Phương thức gọi trực tiếp ‘repository.findById(id)’ và trả về kết quả mà không bổ sung business logic, áp dụng pass-through pattern. Việc sử dụng ‘Optional’ giúp tránh lỗi NullPointerException. Phương thức này có cấu trúc hoàn toàn tương tự ‘BookingService.getBookingById()’, thể hiện sự nhất quán trong thiết kế.

3.4.2.7 Method getAllEmployees(): Lấy Tất cả Employees

```
    /
     Get all employees.
    /
    public List<Employee> getAllEmployees() {
        return repository.findAll();
    }
```

JavaDoc tại các dòng 57–59 mô tả phương thức ‘public’ được gọi từ ‘EmployeeController.loadEmployees()’,’ShiftController.populateEmployeeCombo Box()’ dùng để lấy toàn bộ danh sách employee và trả về ‘List<Employee>‘. Phương thức gọi trực tiếp ‘repository.findAll()’ và trả về kết quả theo pass-through pattern, tương tự ‘BookingService.getAllBookings()’. Việc phương thức này được sử dụng bởi nhiều controller cho thấy service layer có tính tái sử dụng cao, đồng thời đóng vai trò cung cấp dữ liệu tập trung cho các chức năng khác nhau của hệ thống.

3.4.2.8 Method searchEmployeesByName(): Tìm kiếm Employees theo Tên

```
    /
     Search employees by name.
```

```
     /
    public List<Employee> searchEmployeesByName(String name) {
        return repository.findByName(name);
    }
```

JavaDoc tại các dòng 64–66 mô tả phương thức ‘public’ dùng để tìm kiếm employee theo tên, nhận tham số ‘String name’ và trả về ‘List<Employee>‘. Phương thức gọi ‘repository.findByName(name)’ để lọc các employee có tên khớp với tham số (khớp chính xác hoặc một phần tùy theo implementation). Phương thức này tương tự ‘BookingService.searchBookingsByName()’, cung cấp chức năng tìm kiếm và có thể được sử dụng trong tương lai để mở rộng tính năng search.

3.4.2.9 Method validateEmployee(): Validation Dữ liệu

```
    /
      Validate employee data.
     /
    private void validateEmployee(Employee employee) {
        if (employee == null) {
            throw new IllegalArgumentException("Employee cannot be
null");
        }
        if (employee.getName() == null ||
employee.getName().trim().isEmpty()) {
            throw new IllegalArgumentException("Employee name is
required");
        }
        if (employee.getPosition() == null ||
employee.getPosition().trim().isEmpty()) {
            throw new IllegalArgumentException("Employee position is
required");
        }
    }
```

JavaDoc tại các dòng 71–73 mô tả phương thức ‘private’ dùng để validate dữ liệu employee, được gọi từ ‘addEmployee()’ và ‘updateEmployee()’. Việc tách validation thành một helper method giúp tránh lặp code và đảm bảo tính nhất quán. Phương thức trước hết kiểm tra đối tượng employee không ‘null’ để đảm bảo an toàn. Sau đó, các trường bắt buộc được validate, bao gồm ‘name’ và ‘position’, đảm bảo không ‘null’, không rỗng và không chỉ chứa khoảng trắng; nếu không hợp lệ, exception với thông báo rõ ràng sẽ được ném ra. So với ‘BookingService.validateBooking()’, phương thức này chỉ validate hai trường cần thiết, phản ánh mức độ đơn giản hơn của entity ‘Employee’ so với ‘Booking’. Cách tiếp cận này thể hiện pattern minimal validation, trong đó chỉ các field thực sự cần cho business logic mới được kiểm tra, còn các field tùy chọn không bị ràng buộc.

3.4.2.10 Tổng kết EmployeeService

‘EmployeeService’ áp dụng các pattern và nguyên tắc thiết kế tương tự ‘BookingService’, đồng thời được giản lược để phù hợp với đặc thù của entity Employee. Trước hết, đây là một simplified service, chỉ tập trung vào các thao tác CRUD cơ bản, không bao gồm các business operations đặc biệt như hủy hay xác nhận trạng thái. Do đó, logic xử lý và validation cũng đơn giản hơn, chỉ yêu cầu kiểm tra các trường bắt buộc tối thiểu.

Mặc dù đơn giản hơn, ‘EmployeeService’ vẫn duy trì tính nhất quán trong thiết kế khi áp dụng các pattern giống ‘BookingService’, bao gồm Repository Pattern, Validation Layer, Optional Pattern và các pass-through methods cho các thao tác truy vấn. Điều này giúp kiến trúc hệ thống đồng nhất và dễ bảo trì. Service thể hiện rõ tính tái sử dụng, đặc biệt thông qua phương thức ‘getAllEmployees()’, được dùng chung bởi nhiều controller cho các mục đích khác nhau như hiển thị danh sách hoặc populate combo box. Validation được triển khai theo hướng minimal validation, chỉ kiểm tra các field thực sự cần thiết cho nghiệp vụ (name, position), trong khi các field khác được xem là tùy chọn.

Nhìn chung, ‘EmployeeService’ đóng vai trò là lớp bảo vệ dữ liệu và business rules cho domain Employee, với thiết kế gọn nhẹ, nhất quán và phù hợp với một entity có cấu trúc đơn giản hơn so với Booking.

3.4.3 Phân tích InventoryService.java

'InventoryService' là service phức tạp nhất trong hệ thống, chịu trách nhiệm xử lý business logic cho cả inventory items (hàng tồn kho) và inventory transactions (giao dịch nhập/xuất kho). Service này quản lý hai repositories riêng biệt và thực hiện các business operations phức tạp như stock in/out với logic cập nhật quantity và tạo transaction records. Điểm đặc biệt của service này là quản lý mối quan hệ giữa items và transactions, đảm bảo mỗi thao tác stock đều được ghi lại trong transaction history.

3.4.3.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.service;
import com.restaurantmanagement.model.InventoryItem;
import com.restaurantmanagement.model.InventoryTransaction;
import
com.restaurantmanagement.repository.InMemoryInventoryRepository;
import
com.restaurantmanagement.repository.InMemoryInventoryTransactionReposito
ry;
import java.time.LocalDate;
import java.util.List;
import java.util.Optional;
```

Các dòng đầu của 'InventoryService' khai báo package 'com.restaurantmanagement.service'. Service import hai model classes là 'InventoryItem' và 'InventoryTransaction', tương ứng đại diện cho item trong kho và các giao dịch nhập/xuất kho. Đây là điểm khác biệt so với các service khác, khi 'InventoryService' quản lý đồng thời hai entity trong cùng một domain.

Service phụ thuộc vào hai repository riêng biệt là 'InMemoryInventoryRepository' và 'InMemoryInventoryTransactionRepository', phản ánh mức độ phức tạp cao hơn của domain inventory khi cần xử lý cả dữ liệu tồn kho và lịch sử giao dịch. Ngoài ra, 'LocalDate' được import để thiết lập thời điểm

cập nhật cho inventory items và timestamp cho các transactions, cùng với ‘List’ và ‘Optional’ để phục vụ các phương thức truy vấn.

Nhìn chung, các import này cho thấy ‘InventoryService’ có phạm vi trách nhiệm rộng hơn và cấu trúc phức tạp hơn so với các service chỉ quản lý một entity và một repository.

b) Khai báo Class và JavaDoc

```
      /
        Service layer for Inventory business logic.
       /
      public class InventoryService
```

JavaDoc tại các dòng 11–13 mô tả ngắn gọn vai trò của ‘InventoryService’ là service layer chịu trách nhiệm xử lý business logic liên quan đến inventory. Tại dòng 14, class được khai báo với phạm vi truy cập ‘public’, cho phép các lớp ở các package khác, đặc biệt là controller, có thể sử dụng service này.

c) Khai báo Field Repositories

```
      private final InMemoryInventoryRepository itemRepository;
      private final InMemoryInventoryTransactionRepository
transactionRepository;
```

Tại dòng 15, thuộc tính ‘private final InMemoryInventoryRepository itemRepository’ được khai báo để quản lý truy cập dữ liệu cho các inventory items. Tại dòng 16, thuộc tính ‘private final InMemoryInventoryTransactionRepository transactionRepository’ được khai báo để quản lý dữ liệu các inventory transactions. Việc service phụ thuộc vào hai repository riêng biệt là điểm khác biệt so với các service khác, cho thấy ‘InventoryService’ chịu trách nhiệm quản lý hai loại dữ liệu độc lập nhưng có mối quan hệ chặt chẽ với nhau, trong đó các transaction tham chiếu đến inventory items.

3.4.3.2 Constructor và khởi tạo

```
      public InventoryService() {
          this.itemRepository = new InMemoryInventoryRepository();
```

```
        this.transactionRepository = new
InMemoryInventoryTransactionRepository();
    }
```

Constructor tại dòng 18 được khai báo với phạm vi truy cập ‘public’ và không nhận tham số. Tại các dòng 19–20, service khởi tạo trực tiếp hai repository là ‘InMemoryInventoryRepository’ cho inventory items và ‘InMemoryInventoryTransactionRepository’ cho inventory transactions. Việc khởi tạo đồng thời hai repository trong constructor cho thấy ‘InventoryService’ chịu trách nhiệm quản lý cả dữ liệu items và dữ liệu transactions. Đây là điểm khác biệt so với các service khác chỉ làm việc với một repository, phản ánh mức độ phức tạp cao hơn của domain inventory.

3.4.3.3 Method addItem(): Thêm Item mới

```
    /
      Add a new inventory item.
     /
    public InventoryItem addItem(InventoryItem item) {
        validateItem(item);
        item.setLastUpdated(LocalDate.now());
        return itemRepository.save(item);
    }
```

JavaDoc tại các dòng 23–25 mô tả phương thức ‘public’ được gọi từ ‘InventoryController.handleAdd()’, nhận đối tượng ‘InventoryItem’ và trả về ‘InventoryItem’ đã được lưu. Trước khi lưu, phương thức gọi ‘validateItem(item)’ để đảm bảo dữ liệu hợp lệ. Sau đó, service tự động thiết lập trường ‘lastUpdated’ bằng ‘LocalDate.now()’, thể hiện business logic đặc thù của inventory nhằm ghi nhận thời điểm cập nhật cuối cùng của item. Cuối cùng, item được lưu thông qua ‘itemRepository.save(item)’.

So với ‘BookingService.addBooking()’ và ‘EmployeeService.addEmployee()’, phương thức này có thêm xử lý nghiệp vụ là tự động cập nhật ‘lastUpdated’, phản ánh yêu cầu riêng của domain inventory.

3.4.3.4 Method updateItem(): Cập nhật Item

```
      /
        Update an existing inventory item.
       /
      public InventoryItem updateItem(InventoryItem item) {
          if (item.getId() == null || item.getId().isEmpty()) {
              throw new IllegalArgumentException("Item ID is required
for update");
          }
          if (!itemRepository.existsById(item.getId())) {
              throw new IllegalArgumentException("Item with ID " +
item.getId() + " not found");
          }
          validateItem(item);
          item.setLastUpdated(LocalDate.now());
          return itemRepository.save(item);
      }
```

JavaDoc tại các dòng 32–34 mô tả phương thức ‘public’ được gọi từ ‘InventoryController.handleUpdate()’, nhận vào đối tượng ‘InventoryItem’ đã có ID để thực hiện cập nhật. Phương thức trước hết kiểm tra tính hợp lệ của ID, đảm bảo ID không ‘null’ hoặc rỗng; nếu không hợp lệ sẽ ném exception. Tiếp theo, service kiểm tra sự tồn tại của item thông qua ‘itemRepository.existsById(item.getId())’; nếu item không tồn tại, exception sẽ được ném ra. Sau đó, dữ liệu item được validate bằng ‘validateItem(item)’. Trước khi lưu, service tự động cập nhật trường ‘lastUpdated’ bằng ‘LocalDate.now()’, đảm bảo mỗi lần chỉnh sửa đều ghi nhận chính xác thời điểm cập nhật cuối cùng. Cuối cùng, item được lưu thông qua ‘itemRepository.save(item)’. Phương thức này có cấu trúc tương tự các phương thức update ở các service khác, nhưng được mở rộng thêm business logic cập nhật ‘lastUpdated’ date, phản ánh yêu cầu riêng của domain inventory.

3.4.3.5 Method deleteItem(): Xóa Item

```
/
  Delete an inventory item by ID.
 /
public boolean deleteItem(String id) {
    if (id == null || id.isEmpty()) {
        throw new IllegalArgumentException("Item ID cannot be empty");
    }
    return itemRepository.deleteById(id);
}
```

JavaDoc tại các dòng 47–49 mô tả phương thức ‘public’ được gọi từ ‘InventoryController.handleDelete()’, nhận tham số ‘String id’ và trả về ‘boolean’ để biểu thị kết quả xóa. Phương thức thực hiện kiểm tra tính hợp lệ của ID, đảm bảo ID không ‘null’ hoặc rỗng. Sau đó, thao tác xóa được thực hiện thông qua ‘itemRepository.deleteById(id)’ và kết quả được trả về. Cấu trúc phương thức hoàn toàn giống với ‘BookingService.deleteBooking()’ và ‘EmployeeService.deleteEmployee()’, thể hiện pattern thiết kế nhất quán cho các thao tác xóa trong các service.

3.4.3.6 Method getItemById(): Lấy Item theo ID

```
/
  Get inventory item by ID.
 /
public Optional<InventoryItem> getItemById(String id) {
    return itemRepository.findById(id);
}
```

JavaDoc tại các dòng 57–59 mô tả phương thức ‘public’ dùng để truy vấn inventory item theo ID, trả về ‘Optional<InventoryItem>‘ nhằm xử lý an toàn khi item không tồn tại. Phương thức gọi trực tiếp ‘itemRepository.findById(id)’ và trả về kết quả, theo pass-through pattern. Cấu trúc này hoàn toàn giống các service khác và áp dụng Optional pattern để tránh lỗi NullPointerException.

3.4.3.7 Method getAllItems(): Lấy Tất cả Items

```
    /
      Get all inventory items.
     /
    public List<InventoryItem> getAllItems() {
        return itemRepository.findAll();
    }
```

JavaDoc tại các dòng 64–66 mô tả phương thức ‘public’ được gọi từ ‘InventoryController.loadItems()’, trả về ‘List<InventoryItem>‘ chứa toàn bộ items. Phương thức gọi trực tiếp ‘itemRepository.findAll()’ và trả về kết quả theo pass-through pattern. Method này phục vụ việc load tất cả items vào bảng hiển thị trong controller.

3.4.3.8 Method searchItemsByName(): Tìm kiếm Items theo tên

```
    /
      Search items by name.
     /
    public List<InventoryItem> searchItemsByName(String name) {
        return itemRepository.findByName(name);
    }
```

JavaDoc tại các dòng 71–73 mô tả phương thức ‘public’ dùng để tìm kiếm inventory items theo tên, nhận tham số ‘String name’ và trả về ‘List<InventoryItem>‘. Phương thức gọi ‘itemRepository.findByName(name)’ để thực hiện truy vấn. Method này cung cấp chức năng tìm kiếm cho controller và có thể được sử dụng trong tương lai để mở rộng tính năng search.

3.4.3.9 Method getItemsByCategory(): Lấy Items theo danh mục

```
    /
      Get items by category.
     /
    public List<InventoryItem> getItemsByCategory(String category) {
```

```
        return itemRepository.findByCategory(category);
    }
```

JavaDoc tại các dòng 78–80 mô tả phương thức ‘public’ dùng để truy vấn inventory items theo danh mục, nhận tham số ‘String category’ (ví dụ: Food, Beverage, Ingredient) và trả về ‘List<InventoryItem>‘. Phương thức gọi ‘itemRepository.findByCategory(category)’ để thực hiện lọc items theo category. Đây là một query method cho phép filter items, có thể được sử dụng trong tương lai hoặc bởi các controller khác, mặc dù hiện tại ‘InventoryController’ sử dụng ‘filterItems()’ với ‘FilteredList’.

3.4.3.10 Method getLowStockItems(): Lấy Items tồn kho thấp

```
    /
      Get items that are low in stock.
     /
    public List<InventoryItem> getLowStockItems() {
        return itemRepository.findLowStockItems();
    }
```

JavaDoc tại các dòng 85–87 mô tả phương thức ‘public’ dùng để truy vấn các inventory items có tồn kho thấp, trả về ‘List<InventoryItem>‘ chứa các items có ‘quantity < minimumThreshold’. Phương thức gọi ‘itemRepository.findLowStockItems()’, trong đó repository thực hiện lọc items theo điều kiện tồn kho. Đây là một business query hỗ trợ logic nghiệp vụ, giúp xác định các items cần được cảnh báo hoặc bổ sung. Mặc dù controller hiện tại sử dụng ‘item.isLowStock()’ trong bảng, phương thức này có thể được dùng để tạo báo cáo hoặc dashboard.

3.4.3.11 Method stockIn(): Nhập kho

```
    /
      Stock In: Add quantity to an item.
     /
```

```
      public InventoryTransaction stockIn(String itemId, double
quantity, String reason,
                                          String staffId, String
staffName) {
          if (quantity <= 0) {
              throw new IllegalArgumentException("Quantity must be
greater than 0");
          }

          Optional<InventoryItem> itemOpt =
itemRepository.findById(itemId);
          if (!itemOpt.isPresent()) {
              throw new IllegalArgumentException("Item with ID " +
itemId + " not found");
          }

          InventoryItem item = itemOpt.get();
          item.setQuantity(item.getQuantity() + quantity);
          item.setLastUpdated(LocalDate.now());
          itemRepository.save(item);

          // Create transaction record
          InventoryTransaction transaction = new InventoryTransaction();
          transaction.setItemId(itemId);
          transaction.setItemName(item.getName());
          transaction.setQuantity(quantity);
          transaction.setType("IN");
          transaction.setReason(reason);
          transaction.setStaffId(staffId);
          transaction.setStaffName(staffName);
```

```
        return transactionRepository.save(transaction);
    }
```

- Phương thức được khai báo ‘public’ và xử lý nghiệp vụ stock in (nhập kho), nhận các tham số: ‘itemId’, ‘quantity’, ‘reason’, ‘staffId’ và ‘staffName’, trả về đối tượng ‘InventoryTransaction’ đã được tạo. Đây là một thao tác nghiệp vụ phức tạp, không phải CRUD đơn giản.
- Quy trình thực hiện:
    - Validation quantity: Kiểm tra ‘quantity > 0’ để đảm bảo số lượng nhập hợp lệ; sử dụng kiểu ‘double’ để hỗ trợ các giá trị thập phân.
    - Truy vấn item: Lấy item từ repository qua ‘itemRepository.findById(itemId)’ sử dụng ‘Optional’ để xử lý an toàn, ném exception nếu item không tồn tại.
    - Cập nhật item: Cộng số lượng nhập vào tồn kho hiện có (‘item.setQuantity(item.getQuantity() + quantity)’) và cập nhật ‘lastUpdated’ bằng ngày hiện tại (‘LocalDate.now()’).
    - Lưu item đã cập nhật: Gọi ‘itemRepository.save(item)’ để lưu các thay đổi về quantity và lastUpdated.
    - Tạo transaction record: Khởi tạo ‘InventoryTransaction’, gán các thông tin liên quan bao gồm ‘itemId’, ‘itemName’, ‘quantity’, ‘type = "IN"‘, ‘reason’, ‘staffId’ và ‘staffName’.
    - Lưu transaction: Gọi ‘transactionRepository.save(transaction)’ để lưu record và trả về transaction đã tạo.
- Đặc điểm nổi bật:
    - Thực hiện đồng thời cập nhật item và tạo transaction record, đảm bảo tính nhất quán dữ liệu.
    - Hỗ trợ audit trail bằng việc lưu lại thông tin transaction chi tiết.
    - Tuân theo pattern transactional operation, tích hợp validation, cập nhật dữ liệu và ghi nhận lịch sử trong cùng một thao tác nghiệp vụ.

3.4.3.12 Method stockOut(): Xuất Kho

```
    /’
```

```
       * Stock Out: Deduct quantity from an item.
       */
      public InventoryTransaction stockOut(String itemId, double
quantity, String reason,
                                           String staffId, String
staffName) {
          if (quantity <= 0) {
              throw new IllegalArgumentException("Quantity must be
greater than 0");
          }

          Optional<InventoryItem> itemOpt =
itemRepository.findById(itemId);
          if (!itemOpt.isPresent()) {
              throw new IllegalArgumentException("Item with ID " +
itemId + " not found");
          }

          InventoryItem item = itemOpt.get();
          if (item.getQuantity() < quantity) {
              throw new IllegalArgumentException("Insufficient stock.
Available: " +
                      item.getQuantity() + " " + item.getUnit());
          }

          item.setQuantity(item.getQuantity() quantity);
          item.setLastUpdated(LocalDate.now());
          itemRepository.save(item);

          // Create transaction record
          InventoryTransaction transaction = new InventoryTransaction();
```

```
        transaction.setItemId(itemId);
        transaction.setItemName(item.getName());
        transaction.setQuantity(quantity);
        transaction.setType("OUT");
        transaction.setReason(reason);
        transaction.setStaffId(staffId);
        transaction.setStaffName(staffName);

        return transactionRepository.save(transaction);
    }
```

Method ‘stockOut()’ xử lý nghiệp vụ xuất kho, nhận các tham số tương tự ‘stockIn()’ và trả về ‘InventoryTransaction’ vừa tạo. Method thực hiện validation số lượng, truy xuất item từ repository và lấy giá trị từ ‘Optional’. Điểm khác biệt chính so với ‘stockIn()’ là business rule validation: kiểm tra tồn kho có đủ (‘item.getQuantity() >= quantity’) và ném exception nếu không đủ, bao gồm thông tin chi tiết về số lượng hiện có và đơn vị. Sau khi xác nhận hợp lệ, method trừ số lượng xuất khỏi tồn kho, cập nhật ‘lastUpdated’ và lưu item, tiếp đó tạo transaction record với ‘setType("OUT")’ để đánh dấu giao dịch xuất kho và lưu lại transaction. Method này tuân theo cấu trúc của ‘stockIn()’ nhưng bổ sung validation tồn kho và giảm quantity thay vì cộng, thể hiện pattern ‘business rule enforcement’.

3.4.3.13 Method getAllTransactions(): Lấy tất cả Transactions

```
    /’
     * Get all transactions.
     */
    public List<InventoryTransaction> getAllTransactions() {
        return transactionRepository.findAll();
    }
```

Method ‘loadTransactions()’ là phương thức ‘public’ được gọi từ ‘InventoryController.loadTransactions()’ và trả về ‘List<InventoryTransaction>‘ chứa tất cả các transactions. Method thực hiện truy vấn đơn giản thông qua

‘transactionRepository.findAll()’, hoạt động như một pass-through method. Mục đích chính là load toàn bộ transactions vào bảng hiển thị trong controller, tách biệt hoàn toàn với các query liên quan đến items.

3.4.3.14 Method getTransactionsByItemId(): Lấy Transactions theo Item ID

```
    /’
     * Get transactions for a specific item.
     */
    public List<InventoryTransaction> getTransactionsByItemId(String itemId) {
        return transactionRepository.findByItemId(itemId);
    }
```

Method ‘getTransactionsByItemId()’ là phương thức ‘public’ nhận ‘String itemId’ và trả về ‘List<InventoryTransaction>‘ chứa các transactions tương ứng với item đó. Method gọi ‘transactionRepository.findByItemId(itemId)’ để lọc các transaction có ‘itemId’ khớp với tham số, cung cấp lịch sử giao dịch cho một item cụ thể. Mặc dù hiện tại không được sử dụng trong ‘InventoryController’, phương thức này có thể phục vụ việc hiển thị transaction history của từng item hoặc tạo báo cáo.

3.4.3.15 Method validateItem(): Validation dữ liệu

```
    /’
     * Validate inventory item data.
     */
    private void validateItem(InventoryItem item) {
        if (item == null) {
            throw new IllegalArgumentException("Item cannot be null");
        }
        if (item.getName() == null || item.getName().trim().isEmpty())
{
            throw new IllegalArgumentException("Item name is
required");
        }
```

```
        if (item.getCategory() == null ||
item.getCategory().trim().isEmpty()) {
            throw new IllegalArgumentException("Category is
required");
        }
        if (item.getUnit() == null || item.getUnit().trim().isEmpty())
{
            throw new IllegalArgumentException("Unit is required");
        }
        if (item.getQuantity() < 0) {
            throw new IllegalArgumentException("Quantity cannot be
negative");
        }
        if (item.getMinimumThreshold() < 0) {
            throw new IllegalArgumentException("Minimum threshold
cannot be negative");
        }
    }
```

Method ‘validateItem()’ là phương thức ‘private’ helper được gọi từ ‘addItem()’ và ‘updateItem()’ trước khi lưu, nhằm đảm bảo dữ liệu item hợp lệ. Method thực hiện các kiểm tra: item không null; ‘name’, ‘category’, ‘unit’ không null và không rỗng; ‘quantity >= 0’ đảm bảo không âm; ‘minimumThreshold >= 0’ để ngưỡng cảnh báo tồn kho hợp lệ. So với các service khác, validation của ‘InventoryService’ phức tạp hơn ‘EmployeeService’ (chỉ kiểm tra name và position) nhưng ít hơn ‘BookingService’ (validate 8 fields). Phương thức này thể hiện pattern ‘comprehensive validation’, bao quát cả string fields và numeric fields trước khi lưu dữ liệu.

3.4.3.16 Tổng kết InventoryService

‘InventoryService’ là service phức tạp nhất trong hệ thống, thể hiện các pattern và nguyên tắc thiết kế quan trọng trong quản lý domain inventory. Service quản lý

đồng thời cả ‘itemRepository’ và ‘transactionRepository’, xử lý các business operations phức tạp như ‘stockIn()’ và ‘stockOut()’ với việc cập nhật quantity, tạo transaction record và đảm bảo tính nhất quán giữa items và transactions. Business rule enforcement được thể hiện rõ ở ‘stockOut()’, kiểm tra tồn kho trước khi xuất kho. Service còn tự động quản lý metadata (‘lastUpdated’), xử lý dữ liệu numeric cho quantity và threshold, và cung cấp các query methods như ‘getItemsByCategory()’, ‘getLowStockItems()’, ‘getTransactionsByItemId()’ để hỗ trợ các use cases khác nhau. Validation trong service này toàn diện hơn các service khác, bao gồm cả kiểm tra string và numeric fields. Nhìn chung, ‘InventoryService’ đảm bảo tính nhất quán dữ liệu, audit trail đầy đủ cho mọi thay đổi tồn kho, phản ánh sự phức tạp và tính chuyên biệt của domain inventory trong hệ thống nhà hàng.

3.5 Tầng repository

3.5.1 InMemoryBookingRepository

3.5.1.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.repository;

import com.restaurantmanagement.model.Booking;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
```

Đoạn mã này thể hiện vai trò của repository trong kiến trúc phân lớp. Việc khai báo package com.restaurantmanagement.repository xác định rõ namespace và vị trí của lớp trong tầng truy cập dữ liệu. Repository này phụ thuộc trực tiếp vào domain model Booking, thể hiện mối liên kết giữa tầng dữ liệu và đối tượng nghiệp vụ. Các lớp trong Java Collections Framework và Stream API được sử dụng để quản lý và truy vấn dữ liệu trong bộ nhớ, trong đó List đóng vai trò kiểu trả về chuẩn, ArrayList dùng để lưu trữ dữ liệu, và Optional giúp xử lý an toàn các trường hợp không tìm thấy kết quả. Đặc biệt, việc áp dụng Stream API với Collectors cho thấy repository sử dụng phong cách lập trình hàm, giúp các thao tác truy vấn trở nên ngắn gọn, rõ ràng và dễ bảo trì hơn so với cách dùng vòng lặp truyền thống.

b) Khai báo Class và JavaDoc

```
public class InMemoryBookingRepository {  3 usages
    private final List<Booking> bookings;  12 usages
    private int nextId = 1;  1 usage
```

Các dòng JavaDoc mô tả ngắn gọn mục đích của class như một repository lưu trữ dữ liệu trong bộ nhớ (in-memory) cho entity Booking, giúp người đọc nhanh chóng hiểu được vai trò của lớp trong hệ thống. Class được khai báo là public, cho phép các thành phần khác truy cập và sử dụng. Tuy nhiên, class này không implement interface, đồng nghĩa với việc repository được cài đặt trực tiếp mà không có tầng trừu tượng (abstraction layer). Cách tiếp cận này giúp code đơn giản và dễ triển khai, nhưng đồng thời giảm tính linh hoạt, vì trong các kiến trúc phức tạp hơn, việc định nghĩa một interface như BookingRepository sẽ cho phép dễ dàng thay đổi hoặc mở rộng implementation, chẳng hạn chuyển từ lưu trữ trong bộ nhớ sang kết nối cơ sở dữ liệu.

c) Khai báo Field: Danh sách Bookings

Field bookings được khai báo với vai trò là nơi lưu trữ toàn bộ các entity Booking trong bộ nhớ, hoạt động như một "cơ sở dữ liệu" đơn giản của repository. Từ khóa private đảm bảo tính đóng gói, ngăn việc truy cập và thao tác trực tiếp từ bên ngoài, qua đó bảo vệ tính toàn vẹn dữ liệu. Việc sử dụng final giúp cố định tham chiếu của collection sau khi khởi tạo, tăng mức độ an toàn và nhất quán trong thiết kế, dù nội dung bên trong danh sách vẫn có thể được thay đổi thông qua các phương thức của class. Ngoài ra, việc khai báo kiểu List<Booking> thay vì một implementation cụ thể giúp tăng tính linh hoạt, cho phép dễ dàng thay đổi cấu trúc dữ liệu bên dưới trong tương lai mà không ảnh hưởng đến phần còn lại của hệ thống.

d) Khai báo Field: Bộ đếm ID

Field nextId được sử dụng để tạo tự động ID cho các booking mới trong repository. Việc khai báo private đảm bảo field này chỉ được quản lý nội bộ bên trong class, tránh bị can thiệp từ bên ngoài. Kiểu dữ liệu int được khởi tạo với giá trị ban đầu là 1, nhằm đảm bảo booking đầu tiên sẽ có mã định danh BK0001 sau khi được

định dạng. Mỗi khi một booking mới được tạo, giá trị này sẽ được tăng dần (nextId++), từ đó đảm bảo tính duy nhất của ID. Đây là một pattern đơn giản và phổ biến trong các in-memory repository; trong các hệ thống thực tế sử dụng cơ sở dữ liệu, cơ chế này thường được thay thế bằng auto-increment hoặc UUID generator để đảm bảo khả năng mở rộng và an toàn hơn.

3.5.1.2 Constructor và khởi tạo

```
public InMemoryBookingRepository() { this.bookings = new ArrayList<>(); }
```

Constructor không tham số được khai báo public nhằm cho phép tầng service (BookingService) dễ dàng khởi tạo instance của repository. Bên trong constructor, field bookings được khởi tạo bằng một ArrayList rỗng, phù hợp cho việc lưu trữ dữ liệu trong bộ nhớ. Việc lựa chọn ArrayList giúp hỗ trợ truy cập ngẫu nhiên và các thao tác thêm, xóa hiệu quả, đồng thời diamond operator <> (từ Java 7 trở lên) cho phép compiler tự suy luận kiểu generic, giúp code gọn gàng hơn. Collection ban đầu ở trạng thái rỗng và các booking sẽ được thêm vào thông qua phương thức save(). Thiết kế constructor đơn giản này phù hợp với in-memory repository, không đòi hỏi cấu hình phức tạp như trong các repository làm việc với cơ sở dữ liệu, nơi constructor thường phải xử lý kết nối hoặc data source.

3.5.1.3 Method save(): Lưu hoặc Cập nhật Booking

```
public Booking save(Booking booking) {
    if (booking.getId() == null || booking.getId().isEmpty()) {
        // New booking - assign ID
        booking.setId("BK" + String.format("%04d", nextId++));
        bookings.add(booking);
        return booking;
    } else {
        // Update existing booking
        Optional<Booking> existing = findById(booking.getId());
        if (existing.isPresent()) {
            int index = bookings.indexOf(existing.get());
            bookings.set(index, booking);
            return booking;
        } else {
            bookings.add(booking);
            return booking;
        }
    }
}
```

Phương thức save() được khai báo public và được gọi từ tầng service để phục vụ cả chức năng thêm mới và cập nhật booking, thể hiện rõ pattern “save or update” thường gặp trong repository pattern. Phương thức nhận vào một đối tượng Booking

và trả về chính đối tượng đó sau khi đã được xử lý lưu trữ. Trước hết, method kiểm tra điều kiện booking.getId() == null || booking.getId().isEmpty() nhằm xác định booking chưa có ID, qua đảm bảo xử lý an toàn với null và phân biệt rõ trường hợp tạo mới. Với booking mới, hệ thống tự động sinh ID theo định dạng BK0001, BK0002, … bằng cách kết hợp prefix, String.format và biến đếm nextId, sau đó thêm booking vào danh sách và trả về kết quả. Ngược lại, nếu booking đã có ID, repository sẽ tìm đối tượng tương ứng trong danh sách; nếu tồn tại thì cập nhật bằng cách thay thế tại đúng vị trí, nếu không tồn tại thì vẫn thêm vào như một booking mới để tránh mất dữ liệu. Cách thiết kế này giúp đơn giản hóa API khi chỉ cần một phương thức cho cả insert và update, dù trong một số kiến trúc khác có thể tách riêng hai phương thức để tăng tính rõ ràng. Tuy nhiên, cần lưu ý rằng phương thức này chưa đảm bảo thread-safe, do việc tăng nextId và thao tác trên danh sách có thể gây race condition trong môi trường đa luồng; trong hệ thống thực tế, cần các cơ chế đồng bộ phù hợp để khắc phục hạn chế này.

3.5.1.4 Method findById(): Tìm Booking theo ID

```
public Optional<Booking> findById(String id) {
    return bookings.stream()
            .filter( Booking booking -> booking.getId().equals(id))
            .findFirst();
}
```

Phương thức findById() được khai báo public, được sử dụng bởi BookingService.getBookingById() và đồng thời được gọi nội bộ trong phương thức save(). Method nhận vào một String id và trả về Optional<Booking>, thể hiện best practice trong Java 8+ khi xử lý các trường hợp kết quả có thể không tồn tại, thay vì trả về null. Bên trong phương thức, Stream API được sử dụng để duyệt danh sách booking trong bộ nhớ: danh sách được chuyển thành stream, sau đó áp dụng filter với lambda expression để so sánh nội dung ID bằng equals(), và cuối cùng dùng findFirst() để lấy kết quả phù hợp đầu tiên hoặc Optional.empty() nếu không tìm thấy. Cách tiếp cận này giúp code ngắn gọn, rõ ràng và dễ đọc hơn so với vòng lặp truyền thống, đồng thời tăng độ an toàn với null, làm rõ ý nghĩa của method signature và cho phép caller xử lý kết quả linh hoạt thông qua các phương thức của Optional như orElse(), orElseGet() hoặc ifPresent().

3.5.1.5 Method findAll(): Lấy tất cả Bookings

```
public List<Booking> findAll() { return new ArrayList<>(bookings); }
```

Phương thức findAll() được khai báo public và được gọi từ BookingService.getAllBookings() nhằm trả về danh sách toàn bộ các booking hiện có trong repository. Thay vì trả về trực tiếp collection nội bộ, phương thức sử dụng return new ArrayList<>(bookings) để tạo một bản sao (defensive copy) của danh sách. Cách tiếp cận này giúp ngăn chặn việc caller thao tác trực tiếp lên internal state của repository, chẳng hạn như xóa hoặc thêm phần tử vào danh sách gốc, qua đó đảm bảo tính đóng gói và toàn vẹn dữ liệu. Đây là một best practice phổ biến trong repository pattern. Tuy nhiên, cần lưu ý rằng đây chỉ là shallow copy: các đối tượng Booking bên trong danh sách vẫn là cùng một instance, nên nếu caller thay đổi thuộc tính của một booking, dữ liệu trong repository vẫn bị ảnh hưởng; để bảo vệ hoàn toàn, cần áp dụng deep copy cho từng đối tượng.

3.5.1.6 Method deleteById(): Xóa Booking theo ID

```
public boolean deleteById(String id) { return bookings.removeIf( Booking booking -> booking.getId().equals(id)); }
```

Phương thức delete() được khai báo public và được gọi từ BookingService.deleteBooking(), nhận vào một String id và trả về giá trị boolean để biểu thị kết quả của thao tác xóa. Phương thức trả về true nếu tìm thấy booking tương ứng và xóa thành công, hoặc false nếu không tồn tại booking với ID được cung cấp. Việc sử dụng removeIf() của List (từ Java 8+) cho phép xóa các phần tử thỏa mãn điều kiện thông qua một predicate dạng lambda, giúp code ngắn gọn, rõ ràng và an toàn hơn so với cách duyệt và xóa thủ công bằng Iterator. Cách triển khai này sẽ xóa tất cả các booking có ID trùng khớp, tuy nhiên trong điều kiện thiết kế bình thường, ID được đảm bảo là duy nhất, nên thực tế chỉ có tối đa một phần tử bị xóa.

3.5.1.7 Method existsById(): Kiểm tra Booking có tồn tại

```
public boolean existsById(String id) { return bookings.stream().anyMatch( Booking booking -> booking.getId().equals(id)); }
```

Phương thức existsById() được khai báo public và được gọi từ BookingService.updateBooking() nhằm kiểm tra sự tồn tại của booking trước khi

thực hiện cập nhật. Method nhận vào String id và trả về giá trị boolean, trong đó true biểu thị booking tồn tại và false nếu không tìm thấy. Bên trong phương thức, Stream API được sử dụng với anyMatch() để kiểm tra xem có ít nhất một booking trong danh sách có ID khớp hay không. Đây là một short-circuit operation, dừng ngay khi tìm thấy phần tử đầu tiên thỏa điều kiện, giúp tối ưu hiệu năng so với cách tìm kiếm và tạo Optional. Cách tiếp cận này phù hợp cho các trường hợp chỉ cần kiểm tra sự tồn tại, không cần truy xuất toàn bộ đối tượng, qua đó làm cho code gọn gàng và hiệu quả hơn.

3.5.1.8 Method findByDate(): Tìm Bookings theo Ngày

```
public List<Booking> findByDate(java.time.LocalDate date) {  1 usage
    return bookings.stream()
            .filter( Booking booking -> booking.getDate() != null && booking.getDate().equals(date))
            .collect(Collectors.toList());
}
```

Phương thức findByDate() được khai báo public và được gọi từ BookingService.getBookingsByDate(), đóng vai trò là phương thức truy vấn dùng để tìm và lấy toàn bộ các booking theo một ngày cụ thể. Phương thức nhận vào tham số kiểu LocalDate (đối tượng ngày tháng không chứa thông tin thời gian) và trả về danh sách (List<Booking>) các kết quả phù hợp.

Bên trong phương thức, Stream API (API xử lý luồng dữ liệu) được sử dụng để duyệt danh sách booking trong bộ nhớ, kết hợp với điều kiện lọc an toàn với null (null-safe) nhằm tránh phát sinh lỗi NullPointerException khi so sánh ngày. Việc so sánh sử dụng phương thức equals() của LocalDate, đảm bảo đối chiếu chính xác giá trị ngày (năm, tháng, ngày) thay vì so sánh tham chiếu đối tượng. Cuối cùng, các booking thỏa điều kiện được thu thập (collect) lại thành một danh sách mới thông qua Collectors.toList(). Cách cài đặt này giúp mã nguồn ngắn gọn, dễ đọc, an toàn, đồng thời tận dụng hiệu quả API ngày–giờ của Java 8 trong việc xử lý và so sánh dữ liệu theo ngày.

3.5.1.9 Method findByStatus(): Tìm Bookings theo Trạng thái

```
public List<Booking> findByStatus(String status) {  1 usage
    return bookings.stream()
            .filter( Booking booking -> booking.getStatus().equals(status))
            .collect(Collectors.toList());
}
```

Phương thức findByStatus() được khai báo public và được gọi từ BookingService.getBookingsByStatus(), đóng vai trò là phương thức truy vấn dùng để tìm tất cả các booking có trạng thái cụ thể như CONFIRMED, SEATED hoặc CANCELLED. Phương thức nhận vào tham số String status và trả về danh sách (List<Booking>) các booking thỏa điều kiện.

Bên trong phương thức, Stream API (API xử lý luồng dữ liệu) được sử dụng để chuyển danh sách booking thành stream, sau đó áp dụng điều kiện lọc dựa trên việc so sánh trạng thái thông qua phương thức equals(), đảm bảo so sánh nội dung chuỗi thay vì tham chiếu đối tượng. Các booking phù hợp được thu thập (collect) lại thành một danh sách mới bằng Collectors.toList().

Tuy nhiên, phương thức này chưa xử lý an toàn với null, do không kiểm tra booking.getStatus() trước khi gọi equals(), điều này có thể dẫn đến NullPointerException nếu dữ liệu không hợp lệ. Trong môi trường thực tế, cần bổ sung kiểm tra null hoặc sử dụng Objects.equals() để tăng độ an toàn và độ bền vững (robustness) của mã nguồn.

3.5.1.10 Method findByCustomerName(): Tìm Bookings theo Tên Khách hàng

```
    public List<Booking> findByCustomerName(String name) {  1 usage
        String searchName = name.toLowerCase();
        return bookings.stream()
                .filter( Booking booking -> booking.getCustomerName().toLowerCase().contains(searchName))
                .collect(Collectors.toList());
    }
}
```

Phương thức findByCustomerName() được khai báo public và được gọi từ BookingService.searchBookingsByName(), dùng để tìm kiếm booking theo tên khách hàng. JavaDoc mô tả rõ chức năng của phương thức là tìm kiếm theo kiểu không phân biệt chữ hoa – chữ thường (case-insensitive) và khớp một phần (partial match). Phương thức nhận vào tham số String name và trả về danh sách (List<Booking>) các booking thỏa điều kiện.

Bên trong phương thức, chuỗi tìm kiếm được chuyển sang chữ thường và lưu vào một biến cục bộ nhằm tránh việc gọi toLowerCase() lặp lại nhiều lần trong quá trình xử lý, giúp mã nguồn hiệu quả và dễ đọc hơn. Sau đó, Stream API được sử dụng để duyệt danh sách booking trong bộ nhớ, trong đó tên khách hàng cũng được chuyển về chữ thường và so sánh bằng contains() để kiểm tra xem có chứa chuỗi tìm kiếm

hay không. Cách tiếp cận này cho phép tìm kiếm linh hoạt, không yêu cầu khớp chính xác toàn bộ tên, ví dụ khi tìm "nguyen" có thể trả về "Nguyen Van A" hoặc "Tran Nguyen".

Tuy nhiên, phương thức hiện tại chưa xử lý an toàn với giá trị null, do không kiểm tra tham số name hoặc customerName trước khi thao tác. Trong các hệ thống thực tế, cần bổ sung kiểm tra null để tránh phát sinh lỗi và nâng cao độ ổn định (robustness) của chức năng tìm kiếm.

3.5.1.11 Tổng kết InMemoryBookingRepository

Lớp repository này được thiết kế theo Repository Pattern (mẫu kho dữ liệu), đóng vai trò như một tầng trừu tượng giữa logic nghiệp vụ và cơ chế lưu trữ dữ liệu. Việc sử dụng lưu trữ trong bộ nhớ (in-memory) thông qua ArrayList giúp triển khai đơn giản, phù hợp cho giai đoạn prototype và kiểm thử. Các thao tác truy vấn được cài đặt bằng Stream API, làm cho mã nguồn ngắn gọn, dễ đọc và dễ bảo trì. Ngoài ra, phương thức findAll() áp dụng defensive copy (sao chép phòng thủ) để bảo vệ trạng thái nội bộ của repository, tránh việc bị thay đổi trực tiếp từ bên ngoài.

Về điểm mạnh, class có cấu trúc rõ ràng, dễ hiểu, tận dụng các tính năng hiện đại của Java như Stream API, đồng thời cung cấp đầy đủ các phương thức truy vấn đáp ứng những tình huống sử dụng phổ biến như tìm theo ngày, trạng thái và tên khách hàng. Cơ chế tự động sinh ID với định dạng nhất quán cũng giúp quản lý dữ liệu thuận tiện và trực quan.

Tuy nhiên, vẫn tồn tại một số hạn chế cần cải thiện. Trước hết, class này chưa đảm bảo an toàn trong môi trường đa luồng (thread-safe), do việc tăng nextId và thao tác trên ArrayList có thể gây ra lỗi cạnh tranh. Một số phương thức còn thiếu kiểm tra null, tiềm ẩn nguy cơ phát sinh lỗi khi dữ liệu không hợp lệ. Bên cạnh đó, việc không định nghĩa interface cho repository làm giảm tính linh hoạt khi cần thay đổi hoặc mở rộng cách lưu trữ, chẳng hạn chuyển từ in-memory sang cơ sở dữ liệu. Cơ chế sinh ID hiện tại cũng còn đơn giản và chưa phù hợp cho hệ thống lớn, đồng thời repository không hỗ trợ transaction, nên không thể hoàn tác khi xảy ra lỗi.

Trong biểu đồ UML, lớp này tương ứng với InMemoryBookingRepository, với các phương thức được mô tả trực tiếp trong sơ đồ. Quan hệ BookingService →

InMemoryBookingRepository (uses) được thể hiện thông qua việc tầng service khởi tạo và gọi các phương thức của repository để thực hiện các thao tác nghiệp vụ liên quan đến booking.

3.5.2 InMemoryEmployeeRepository

3.5.2.1 Cấu trúc Class và Khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.repository;

import com.restaurantmanagement.model.Employee;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
```

So với Package và Import Statements của Booking repository, phần khai báo này không có khác biệt về cấu trúc hay cách tổ chức, mà chỉ khác ở đối tượng nghiệp vụ được quản lý. Cụ thể, repository này phụ thuộc vào domain model Employee thay vì Booking, trong khi các import từ Java Collections Framework và API xử lý luồng dữ liệu (Stream API) được giữ nguyên, thể hiện tính nhất quán trong thiết kế các repository của hệ thống.

b) Khai báo Class và JavaDoc

```
public class InMemoryEmployeeRepository {  no usages
    private final List<Employee> employees = new ArrayList();
    private int nextId = 1;
```

So với InMemoryBookingRepository, phần khai báo package và import của repository này giữ nguyên cấu trúc và cách tổ chức, chỉ khác ở đối tượng nghiệp vụ được quản lý là Employee thay vì Booking. Các import từ Java Collections Framework và API xử lý luồng dữ liệu (Stream API) được sử dụng tương tự, thể hiện tính đồng bộ và nhất quán trong thiết kế repository của hệ thống.

c) Khai báo Field: Danh sách Employees

So với field bookings trong Booking repository, field employees không khác về mặt cấu trúc hay cách thiết kế, mà chỉ khác ở đối tượng nghiệp vụ được lưu trữ là Employee thay vì Booking.

d) Khai báo Field: Bộ đếm ID

Field nextId được sử dụng để tự động sinh mã định danh (ID) cho các đối tượng nhân viên (Employee) mới. So với InMemoryBookingRepository, điểm khác biệt nằm ở tiền tố của ID, trong đó "EMP" được sử dụng thay cho "BK", giúp phân biệt rõ ràng loại đối tượng nghiệp vụ và đảm bảo tính nhất quán, dễ nhận diện trong toàn bộ hệ thống.

3.5.2.2 Method save(): Lưu hoặc Cập nhật Employee

```
public Employee save(Employee employee) {
    if (employee.getId() != null && !employee.getId().isEmpty()) {
        Optional<Employee> existing = this.findById(employee.getId());
        if (existing.isPresent()) {
            int index = this.employees.indexOf(existing.get());
            this.employees.set(index, employee);
            return employee;
        } else {
            this.employees.add(employee);
            return employee;
        }
    } else {
        Object[] var10002 = new Object[]{this.nextId++};
        employee.setId("EMP" + String.format("%04d", var10002));
        this.employees.add(employee);
        return employee;
    }
}
```

Về mặt thiết kế, phương thức này hoàn toàn tương đồng với InMemoryBookingRepository.save() về cấu trúc và logic xử lý, chỉ khác ở tiền tố ID (EMP thay vì BK) và loại đối tượng nghiệp vụ.

3.5.2.3 Method findById(): Tìm Employee theo ID

```
public Optional<Employee> findById(String id) {
    return this.employees.stream().filter(( Employee emp) -> emp.getId().equals(id)).findFirst();
}
```

Về cấu trúc và logic, phương thức này hoàn toàn tương đồng với InMemoryBookingRepository.findById(), chỉ khác ở tên biến trong biểu thức lambda và loại đối tượng nghiệp vụ được xử lý.

3.5.2.4 Method findAll(): Lấy Tất cả Employees

```
public List<Employee> findAll() { return new ArrayList(this.employees); }
```

Phương thức findAll() của Employee repository có cách cài đặt hoàn toàn giống với InMemoryBookingRepository.findAll(), sử dụng sao chép phòng thủ (defensive copy) để trả về danh sách nhân viên mà không làm lộ collection nội bộ. Cách tiếp cận này đảm bảo tính đóng gói và toàn vẹn dữ liệu, dù các đối tượng Employee bên trong vẫn là sao chép nông (shallow copy).

3.5.2.5 Method deleteById(): Xóa Employee theo ID

```
public boolean deleteById(String id) { return this.employees.removeIf(( Employee emp) -> emp.getId().equals(id)); }
```

Phương thức deleteById() của Employee repository được cài đặt hoàn toàn tương đồng với InMemoryBookingRepository.deleteById(), sử dụng removeIf() để xóa nhân viên theo mã định danh (ID).

3.5.2.6 Method existsById(): Kiểm tra Employee có Tồn tại

```
public boolean existsById(String id) { return this.employees.stream().anyMatch(( Employee emp) -> emp.getId().equals(id)); }
```

Phương thức existsById() của Employee repository có cách cài đặt hoàn toàn giống với InMemoryBookingRepository.existsById(), sử dụng API xử lý luồng dữ liệu (Stream API) cùng với anyMatch() để kiểm tra sự tồn tại của nhân viên theo ID.

3.5.2.7 Method findByName(): Tìm Employees theo Tên

```
    public List<Employee> findByName(String name) {
        return (List)this.employees.stream().filter(( Employee emp) -> emp.getName().toLowerCase().contains(name.toLowerCase())).collect(Collectors.toList());
    }
}
```

Phương thức findByName() của Employee repository dùng để tìm kiếm nhân viên theo tên với tiêu chí không phân biệt chữ hoa – chữ thường và khớp một phần, thông qua Stream API kết hợp filter() và collect(). Cách cài đặt này cho phép so sánh linh hoạt bằng contains() sau khi chuyển cả tên nhân viên và chuỗi tìm kiếm về chữ thường, nhờ đó có thể tìm được các kết quả như “John”, “JOHN” hay “johnny” khi tìm “john”.

3.5.2.8 Tổng kết InMemoryEmployeeRepository

Các phần giống nhau với InMemoryBookingRepository là cấu trúc chung của repository: đều sử dụng mẫu kho lưu trữ (repository pattern) với lớp trừu tượng (abstraction layer) giữa logic nghiệp vụ và lưu trữ dữ liệu, lưu trữ trong bộ nhớ bằng ArrayList, sử dụng API dòng (Stream API) cho các truy vấn, áp dụng sao chép phòng ngừa (defensive copy) trong findAll(), và có các phương thức CRUD như lưu hoặc cập nhật (save or update), kiểm tra tồn tại theo ID (existsById), xóa theo ID (deleteById). Ngoài ra, Repository nhân viên (InMemoryEmployeeRepository) sử dụng tiền tố ID "EMP" thay vì "BK", và chỉ có một phương thức truy vấn (query method) là findByName() (thay vì nhiều phương thức truy vấn như findByDate(), findByStatus(), findByCustomerName() trong booking). Phương thức findByName() đơn giản hơn nhưng có thể tối ưu bằng cách dùng biến cục bộ (local variable) để tránh gọi toLowerCase() nhiều lần, giúp hiệu năng tốt hơn.

3.5.3 InMemoryInventoryRepository

3.5.3.1 Cấu trúc Class và Khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.repository;

import com.restaurantmanagement.model.InventoryItem;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
```

Phần khai báo package và import của repository này được tổ chức tương tự các repository khác trong hệ thống. Điểm khác biệt duy nhất là repository phụ thuộc vào domain model InventoryItem thay vì các entity khác. Các import từ Java Collections Framework và Stream API được giữ nguyên, cho thấy thiết kế repository được xây dựng đồng bộ và nhất quán, phục vụ cho việc lưu trữ và truy vấn dữ liệu trong bộ nhớ.

b) Khai báo Class và JavaDoc

```
public class InMemoryInventoryRepository {  3 usages
    private final List<InventoryItem> items;  12 usages
    private int nextId = 1;  1 usage
```

Class InMemoryInventoryRepository được khai báo public kèm theo JavaDoc mô tả rõ đây là repository lưu trữ InventoryItem trong bộ nhớ. Cách khai báo này hoàn toàn nhất quán với các repository khác trong hệ thống, khi class được cài đặt trực tiếp mà chưa thông qua interface trừu tượng, phù hợp cho mô hình đơn giản và dễ mở rộng trong tương lai nếu cần thêm abstraction layer.

c) Khai báo Field: Danh sách Items

So với field bookings trong Booking repository, field items trong Inventory repository không khác về mặt cấu trúc hay cách thiết kế, mà chỉ khác ở đối tượng nghiệp vụ được lưu trữ là InventoryItem thay vì Booking.

d) Khai báo Field: Bộ đếm ID

Field nextId trong Inventory repository được sử dụng để tự động tạo ID cho các inventory item mới giống như trong Booking repository. Điểm khác biệt là giá trị khởi tạo được gán là 1 để ID đầu tiên có dạng "INV0001", đồng thời prefix của ID là "INV" thay vì "BK" hay "EMP".

3.5.3.2 Constructor và Khởi tạo

```
public InMemoryInventoryRepository() { this.items = new ArrayList<>(); }
```

Constructor InMemoryInventoryRepository() là public và không tham số, khởi tạo items bằng new ArrayList<>() để lưu trữ các inventory item, tương tự như constructor trong InMemoryBookingRepository() với bookings.

3.5.3.3 Method save(): Lưu hoặc Cập nhật Item

```
public InventoryItem save(InventoryItem item) {
    if (item.getId() == null || item.getId().isEmpty()) {
        // New item - assign ID
        item.setId("INV" + String.format("%04d", nextId++));
        items.add(item);
        return item;
    } else {
        // Update existing item
        Optional<InventoryItem> existing = findById(item.getId());
        if (existing.isPresent()) {
            int index = items.indexOf(existing.get());
            items.set(index, item);
            return item;
        } else {
            items.add(item);
            return item;
        }
    }
}
```

Phương thức này hoàn toàn giống các repository khác về cấu trúc và logic, chỉ khác ở tiền tố ID ("INV" thay vì "BK" hoặc "EMP") và loại đối tượng nghiệp vụ được lưu trữ.

3.5.3.4 Method findById(): Tìm Item theo ID

```
public Optional<InventoryItem> findById(String id) {
    return items.stream()
            .filter( InventoryItem item -> item.getId().equals(id))
            .findFirst();
}
```

Phương thức getItemById(String id) là public và trả về Optional<InventoryItem>, được gọi từ InventoryService và sử dụng nội bộ trong save(). Phương thức này về cấu trúc và logic hoàn toàn giống các repository khác, nhưng khác ở đối tượng nghiệp vụ là InventoryItem.

3.5.3.5 Method findAll(): Lấy Tất cả Items

```
public List<InventoryItem> findAll() { return new ArrayList<>(items); }
```

Phương thức findAll() là public và trả về List<InventoryItem> chứa tất cả các item, được gọi từ InventoryService.getAllItems(). Cấu trúc và logic của phương thức này giống các repository khác.

3.5.3.6 Method deleteById(): Xóa Item theo ID

```
public boolean deleteById(String id) { return items.removeIf( InventoryItem item -> item.getId().equals(id)); }
```

Phương thức deleteItem(String id) là public và được gọi từ InventoryService.deleteItem(), trả về boolean cho biết việc xóa thành công hay không. Cấu trúc và logic của phương thức này giống các repository khác.

3.5.3.7 Method existsById(): Kiểm tra Item có Tồn tại

```
public boolean existsById(String id) { return items.stream().anyMatch( InventoryItem item -> item.getId().equals(id)); }
```

Phương thức existsById(String id) là công khai và được gọi từ InventoryService.updateItem() để kiểm tra sự tồn tại của item trước khi cập nhật, trả về giá trị boolean. Cấu trúc và logic của phương thức này giống các repository khác.

3.5.3.8 Method findByName(): Tìm Items theo Tên

```
public List<InventoryItem> findByName(String name) {  1 usage
    String searchName = name.toLowerCase();
    return items.stream()
            .filter( InventoryItem item -> item.getName().toLowerCase().contains(searchName))
            .collect(Collectors.toList());
}
```

Phương thức findByName(String name) là công khai và được gọi từ InventoryService.searchItemsByName(), trả về danh sách InventoryItem có tên khớp với chuỗi tìm kiếm theo kiểu không phân biệt hoa thường và khớp một phần. Chuỗi tìm kiếm được chuyển sang chữ thường (searchName = name.toLowerCase()) để so sánh không phân biệt hoa thường, đồng thời lưu trong biến cục bộ nhằm tối ưu hóa và tránh gọi toLowerCase() nhiều lần trong Stream. Phương thức này sử dụng Stream API để lọc các item: items.stream() chuyển list thành luồng, .filter(item -> item.getName().toLowerCase().contains(searchName)) kiểm tra xem tên item có chứa chuỗi tìm kiếm hay không, và .collect(Collectors.toList()) gom các phần tử khớp thành danh sách trả về. So với InMemoryEmployeeRepository.findByName(), phương thức này tối ưu hơn nhờ sử dụng biến cục bộ searchName. Tuy nhiên, vẫn cần lưu ý rằng các thuộc tính name hoặc item.getName() chưa được kiểm tra null; trong môi trường thực tế nên bổ sung kiểm tra null để đảm bảo an toàn.

3.5.3.9 Method findByCategory(): Tìm Items theo Danh mục

```
public List<InventoryItem> findByCategory(String category) {  1 usage
    return items.stream()
            .filter( InventoryItem item -> item.getCategory().equals(category))
            .collect(Collectors.toList());
}
```

Phương thức getItemsByCategory(String category) là công khai và được gọi từ InventoryService.getItemsByCategory(), trả về danh sách InventoryItem có danh mục khớp chính xác với tham số category. Đây là một query method để lọc các item theo danh mục cố định, ví dụ: "Food", "Beverage", "Ingredient", "Cleaning" hoặc "Others". Phương thức này sử dụng Stream API: items.stream() chuyển list thành luồng, .filter(item -> item.getCategory().equals(category)) lọc các item có danh mục khớp chính xác (exact match), và .collect(Collectors.toList()) gom các phần tử đã lọc thành danh sách trả về. Khác với findByName() sử dụng partial match với contains(), phương thức này sử dụng exact match với equals(), phù hợp vì danh mục là giá trị cố định.

3.5.3.10 Method findLowStockItems(): Tìm Items Tồn kho Thấp

```
    public List<InventoryItem> findLowStockItems() {  1 usage
        return items.stream()
                .filter(InventoryItem::isLowStock)
                .collect(Collectors.toList());
    }
}
```

Phương thức getLowStockItems() là công khai và được gọi từ InventoryService.getLowStockItems(), không nhận tham số và trả về danh sách InventoryItem có tồn kho thấp (số lượng ≤ ngưỡng tối thiểu). Đây là một phương thức truy vấn đặc biệt, sử dụng API luồng dữ liệu (Stream API) với tham chiếu phương thức InventoryItem::isLowStock để lọc các item có tồn kho thấp. Tham chiếu phương thức này là cú pháp rút gọn của biểu thức lambda item -> item.isLowStock(), giúp mã nguồn ngắn gọn và dễ đọc hơn. Phương thức isLowStock() là logic nghiệp vụ của lớp InventoryItem, kiểm tra xem số lượng tồn kho có nhỏ hơn hoặc bằng ngưỡng tối thiểu hay không.

3.5.3.11 Tổng kết InMemoryInventoryRepository

Repository này có cấu trúc CRUD tương tự các repository khác, sử dụng Stream API cho các thao tác truy vấn, áp dụng sao chép phòng thủ (defensive copy) trong phương thức findAll(), và có các phương thức lưu hoặc cập nhật (save or update), tồn tại theo ID (existsById()), xóa theo ID (deleteById()) tương tự, thể hiện

tính nhất quán trong thiết kế. Tuy nhiên, Repository này sử dụng tiền tố ID "INV" cho đối tượng InventoryItem, có nhiều phương thức truy vấn hơn (tìm theo tên (findByName), tìm theo danh mục (findByCategory), tìm các mặt hàng tồn kho thấp (findLowStockItems)) so với 1-3 phương thức thông thường; đặc biệt, findLowStockItems() sử dụng tham chiếu phương thức (method reference) và ủy quyền logic nghiệp vụ cho domain model (isLowStock), thể hiện mô hình "rich domain model". Ngoài ra, findByName() được tối ưu hơn so với InMemoryEmployeeRepository nhờ sử dụng biến cục bộ để giảm lặp lại gọi getter.

3.5.4 InMemoryInventoryTransactionRepository

3.5.4.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.repository;

import com.restaurantmanagement.model.InventoryTransaction;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
```

Repository này thuộc gói com.restaurantmanagement.repository và quản lý thực thể InventoryTransaction từ domain model. Cách khai báo package và import này giống với các repository khác trong hệ thống.

b) Khai báo Class và JavaDoc

```
public class InMemoryInventoryTransactionRepository {  no usages
    private final List<InventoryTransaction> transactions = new ArrayList();
    private int nextId = 1;
```

Lớp InMemoryInventoryTransactionRepository là cài đặt repository trong bộ nhớ cho các thực thể InventoryTransaction. Đây là lớp công khai (public), cho phép các lớp khác truy cập trực tiếp. Tương tự các repository khác, lớp này không triển khai giao diện (interface) nào, nghĩa là repository được implement trực tiếp mà không có lớp trừu tượng (abstraction layer).

c) Khai báo Field: Danh sách Transactions

Field transactions được khai báo là private final List<InventoryTransaction>, tương tự các repository khác. Collection này lưu trữ tất cả các giao dịch tồn kho trong bộ nhớ, đóng vai trò như một cơ sở dữ liệu tạm thời (in-memory database).

d) Khai báo Field: Bộ đếm ID

Field nextId được sử dụng để tự động tạo ID cho các giao dịch tồn kho mới. Nó được khai báo là private int, giá trị khởi tạo là 1 để đảm bảo ID đầu tiên có dạng "TXN0001". Mỗi khi tạo transaction mới, nextId được tăng dần (nextId++) để đảm bảo tính duy nhất của ID.

3.5.4.2 Method save(): Lưu Transaction Mới

```
public InventoryTransaction save(InventoryTransaction transaction) {
    if (transaction.getId() == null || transaction.getId().isEmpty()) {
        Object[] var10002 = new Object[]{this.nextId++};
        transaction.setId("TXN" + String.format("%04d", var10002));
    }

    this.transactions.add(transaction);
    return transaction;
}
```

Phương thức save(InventoryTransaction transaction) có chức năng lưu một giao dịch tồn kho mới vào repository. Khi được gọi từ InventoryService.stockIn() hoặc InventoryService.stockOut(), phương thức nhận một đối tượng giao dịch làm tham số và trả về giao dịch đã được lưu.

Logic hoạt động như sau: trước tiên, phương thức kiểm tra xem giao dịch đã có ID hay chưa (transaction.getId() == null || transaction.getId().isEmpty()). Nếu chưa có, đây là giao dịch mới, phương thức sẽ tự động tạo ID theo định dạng "TXN0001", "TXN0002",… bằng cách kết hợp tiền tố "TXN" với số thứ tự 4 chữ số (String.format("%04d", nextId++)). Cách làm này đảm bảo mỗi giao dịch có ID duy nhất và tăng dần.

Sau khi đảm bảo giao dịch có ID, phương thức sẽ thêm giao dịch vào danh sách transactions. Phương thức này không thực hiện bất kỳ logic cập nhật nào cho các giao dịch đã tồn tại, phù hợp với mẫu "chỉ thêm mới" (append-only), vì các giao dịch được coi là bản ghi kiểm toán bất biến.

3.5.4.3 Method findById(): Tìm Transaction theo ID

```
public Optional<InventoryTransaction> findById(String id) {
    return this.transactions.stream().filter(( InventoryTransaction txn) -> txn.getId().equals(id)).findFirst();
}
```

Phương thức findById(String id) về cấu trúc giống các repository khác, sử dụng Stream API và Optional để xử lý null-safe. Điểm khác là đối tượng thao tác là InventoryTransaction và biến lambda sử dụng tên txn thay vì item hay booking, phản ánh rõ kiểu thực thể đang quản lý.

3.5.4.4 Method findAll(): Lấy Tất cả Transactions

```
public List<InventoryTransaction> findAll() { return new ArrayList(this.transactions); }
```

Phương thức getAllTransactions() trả về danh sách các giao dịch tồn kho dưới dạng bản sao của collection nội bộ (defensive copy), đảm bảo caller không thể thay đổi trực tiếp trạng thái bên trong repository. Điểm khác so với các repository khác là đối tượng trong danh sách là InventoryTransaction. Còn lại, logic về defensive copy và bảo vệ trạng thái nội bộ giống hệt, thể hiện tính nhất quán trong thiết kế.

3.5.4.5 Method findByItemId(): Tìm Transactions theo Item ID

```
public List<InventoryTransaction> findByItemId(String itemId) {
    return (List)this.transactions.stream().filter(( InventoryTransaction txn) -> txn.getItemId().equals(itemId)).collect(Collectors.toList());
}
```

Phương thức getTransactionsByItemId(String itemId) được sử dụng để tìm tất cả các giao dịch tồn kho liên quan đến một mặt hàng cụ thể. Khi được gọi, phương thức nhận tham số itemId và chuyển danh sách các giao dịch (transactions) thành một luồng dữ liệu (stream) bằng transactions.stream(). Sau đó, phương thức lọc các giao dịch sao cho chỉ giữ lại những phần tử có txn.getItemId() bằng chính xác với itemId thông qua biểu thức lambda txn -> txn.getItemId().equals(itemId). Cuối cùng, các giao dịch đã lọc được thu thập thành một danh sách mới bằng .collect(Collectors.toList()) và trả về cho caller.

3.5.4.6 Method findByType(): Tìm Transactions theo Loại

```
public List<InventoryTransaction> findByType(String type) {
    return (List)this.transactions.stream().filter(( InventoryTransaction txn) -> txn.getType().equals(type)).collect(Collectors.toList());
}
```

Phương thức getTransactionsByType(String type) được sử dụng để tìm các giao dịch tồn kho theo loại, thường là "IN" cho nhập kho hoặc "OUT" cho xuất kho. Khi được gọi, phương thức nhận tham số type và chuyển danh sách các giao dịch (transactions) thành một luồng dữ liệu (stream) bằng transactions.stream(). Tiếp theo, phương thức lọc các giao dịch sao cho chỉ giữ lại những phần tử có txn.getType() bằng chính xác với type thông qua biểu thức lambda txn -> txn.getType().equals(type). Cuối cùng, các giao dịch đã lọc được thu thập thành một danh sách mới bằng .collect(Collectors.toList()) và trả về cho caller.

3.5.4.7 Method findByStaffId(): Tìm Transactions theo Staff ID

```
public List<InventoryTransaction> findByStaffId(String staffId) {
    return (List)this.transactions.stream().filter(( InventoryTransaction txn) -> txn.getStaffId() != null && txn.getStaffId().equals(staffId)).collect(Collectors.toList());
}
```

Phương thức getTransactionsByStaffId(String staffId) được sử dụng để tìm tất cả các giao dịch tồn kho do một nhân viên cụ thể thực hiện. Khi được gọi, phương thức nhận tham số staffId và chuyển danh sách các giao dịch (transactions) thành luồng dữ liệu bằng transactions.stream(). Trong quá trình lọc, phương thức kiểm tra giá trị null trước khi so sánh txn.getStaffId() != null && txn.getStaffId().equals(staffId), đảm bảo không xảy ra lỗi khi một số giao dịch không có thông tin nhân viên (ví dụ giao dịch tự động). Các giao dịch có mã nhân viên (staffId) khớp chính xác với tham số sẽ được thu thập thành một danh sách mới bằng .collect(Collectors.toList()) và trả về cho caller (người gọi phương thức).

3.5.4.8 Tổng kết InMemoryInventoryTransactionRepository

Repository InMemoryInventoryTransaction có các phương thức truy vấn (findById(), findAll()) và lưu dữ liệu trong bộ nhớ (ArrayList) giống các repository khác, sử dụng Stream API cho các thao tác lọc, và trả về defensive copy để bảo vệ

internal state. Điểm khác biệt quan trọng là Append-Only Pattern: phương thức save() chỉ thêm transactions mới, không có logic cập nhật hay xóa, phù hợp với immutable transaction pattern, vì transactions được coi là audit trail không nên bị sửa đổi. Prefix ID "TXN" cũng riêng biệt so với các repository khác (ví dụ "BK" cho Booking, "EMP" cho Employee), đảm bảo định dạng ID nhất quán. Repository có nhiều query methods đặc thù hơn để hỗ trợ các use case khác nhau: findByItemId() cho lịch sử item, findByType() phân loại IN/OUT, findByStaffId() theo dõi nhân viên với null check tốt hơn, vì staffId có thể là optional field. Pattern append-only này đảm bảo tính toàn vẹn của audit trail: nếu cần "sửa" một transaction, phải tạo transaction mới (adjustment transaction) thay vì sửa trực tiếp. Điều này khác với các repository CRUD thông thường, nơi có thể có update hoặc delete methods.

3.5.5 InMemoryShiftRepository

3.5.5.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.repository;

import com.restaurantmanagement.model.Shift;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
```

Repository InMemoryShiftRepository được đặt trong package com.restaurantmanagement.repository, giống như các repository khác và phù hợp với cấu trúc package theo kiến trúc phân lớp. Repository này quản lý entity Shift, do đó import trực tiếp lớp com.restaurantmanagement.model.Shift như một dependency vào domain model. Khác với các repository chỉ import từ java.util, repository này còn import LocalDate từ java.time để xử lý các phương thức truy vấn theo ngày như findByDate() và findByDateRange().

b) Khai báo Class và JavaDoc

```
public class InMemoryShiftRepository {  no usages
    private final List<Shift> shifts = new ArrayList();
    private int nextId = 1;
```

Class InMemoryShiftRepository được mô tả trong JavaDoc là một implementation của repository lưu trữ dữ liệu trong bộ nhớ cho entity Shift, với ghi chú rằng sử dụng ArrayList để lưu trữ dữ liệu, tương tự như InMemoryEmployeeRepository.

c) Khai báo Field: Danh sách Shifts

Field shifts được khai báo là private final List<Shift>, tương tự các repository khác, đóng vai trò lưu trữ tất cả các shift trong bộ nhớ như một "database" tạm thời. Việc sử dụng final đảm bảo tham chiếu không thay đổi, trong khi nội dung bên trong vẫn có thể thêm, xóa hoặc sửa đổi thông qua các phương thức của class.

d) Khai báo Field: Bộ đếm ID

Field nextId được khai báo là private int và khởi tạo bằng 1 để tự động tạo ID duy nhất cho các shift mới với định dạng "SHF0001", "SHF0002"… Mỗi khi tạo shift mới, nextId sẽ tăng dần (nextId++). Điểm khác biệt so với các repository khác là prefix ID "SHF", phù hợp với entity Shift.

3.5.5.2 Method save(): Lưu hoặc Cập nhật Shift

```
public Shift save(Shift shift) {
    if (shift.getId() != null && !shift.getId().isEmpty()) {
        Optional<Shift> existing = this.findById(shift.getId());
        if (existing.isPresent()) {
            int index = this.shifts.indexOf(existing.get());
            this.shifts.set(index, shift);
            return shift;
        } else {
            this.shifts.add(shift);
            return shift;
        }
    } else {
        Object[] var10002 = new Object[]{this.nextId++};
        shift.setId("SHF" + String.format("%04d", var10002));
        this.shifts.add(shift);
        return shift;
    }
}
```

Phương thức save(Shift shift) thực hiện tạo mới hoặc cập nhật shift, tương tự các repository khác. Điểm khác biệt là prefix ID "SHF" và logic xử lý ID cho shift mới, trong khi cơ chế lưu, cập nhật theo index trong danh sách giống hầu hết các repository khác.

3.5.5.3 Method findById(): Tìm Shift theo ID

```
public Optional<Shift> findById(String id) {
    return this.shifts.stream().filter((Shift shift) -> shift.getId().equals(id)).findFirst();
}
```

Phương thức findById(String id) sử dụng Stream API để tìm shift theo ID và trả về Optional để tránh null, tương tự các repository khác. Điểm khác biệt duy nhất là đối tượng được tìm là Shift, còn cơ chế lọc, null-safe và cách triển khai giống hầu hết các repository khác.

3.5.5.4 Method findAll(): Lấy Tất cả Shifts

```
public List<Shift> findAll() { return new ArrayList(this.shifts); }
```

Phương thức findAll() trả về bản sao (defensive copy) của danh sách shifts để bảo vệ trạng thái nội bộ, cơ chế này giống hầu hết các repository khác. Điểm khác biệt duy nhất là đối tượng trong danh sách là Shift, còn cách triển khai, bảo toàn dữ liệu và đóng gói tương tự các repository khác.

3.5.5.5 Method deleteById(): Xóa Shift theo ID

```
public boolean deleteById(String id) { return this.shifts.removeIf(( Shift shift) -> shift.getId().equals(id)); }
```

Phương thức hoạt động giống deleteById() trong repository Booking: nhận ID và xóa shift khớp khỏi danh sách, trả về true nếu có phần tử bị xóa, false nếu không tìm thấy. Điểm khác biệt duy nhất là entity và prefix ID, còn logic, cấu trúc và cách dùng removeIf() hoàn toàn tương tự.

3.5.5.6 Method existsById(): Kiểm tra Shift có Tồn tại

```
public boolean existsById(String id) { return this.shifts.stream().anyMatch(( Shift shift) -> shift.getId().equals(id)); }
```

Phương thức existsById() trong Shift repository hoạt động giống các repository khác: nhận ID và kiểm tra sự tồn tại của shift trong danh sách bằng anyMatch(). Điểm khác biệt duy nhất là entity cụ thể; logic stream và short-circuit operation hoàn toàn tương tự, trả về true nếu tìm thấy, false nếu không.

3.5.5.7 Method findByEmployeeId(): Tìm Shifts theo Employee ID

```
public List<Shift> findByEmployeeId(String employeeId) {
    return (List)this.shifts.stream().filter(( Shift shift) -> shift.getEmployeeId().equals(employeeId)).collect(Collectors.toList());
}
```

Phương thức getShiftsByEmployeeId() hoạt động giống các query method khác: nhận employeeId và lọc danh sách shifts bằng Stream API, so sánh chuỗi với equals(), trả về danh sách các shift khớp. Điểm khác là filter theo employeeId còn logic stream và collect hoàn toàn tương tự các repository khác.

3.5.5.8 Method findByDate(): Tìm Shifts theo Ngày

```
public List<Shift> findByDate(LocalDate date) {
    return (List)this.shifts.stream().filter(( Shift shift) -> shift.getDate().equals(date)).collect(Collectors.toList());
}
```

Phương thức này lọc danh sách shifts bằng Stream API, so sánh ngày với ngày của LocalDate và trả về danh sách các shift khớp. Cấu trúc, logic stream và collect giống các query method khác như findByDate() trong repository Booking.

3.5.5.9 Method findByDateRange(): Tìm Shifts theo Khoảng Ngày

```
    public List<Shift> findByDateRange(LocalDate startDate, LocalDate endDate) {
        return (List)this.shifts.stream().filter((Shift shift) -> !shift.getDate().isBefore(startDate) && !shift.getDate().isAfter(endDate)).collect(Collectors.toList());
    }
}
```

Phương thức này lọc danh sách shifts bằng Stream API dựa trên khoảng ngày từ startDate đến endDate, bao gồm cả hai ngày biên. Cấu trúc filter và collect giống các query method khác như findByDate() trong repository Booking, nhưng khác ở chỗ đây là kiểm tra khoảng ngày (date range) thay vì chỉ so sánh một ngày duy nhất.

3.5.5.10 Tổng kết InMemoryShiftRepository

Repository Shift có các phương thức CRUD cơ bản giống các repository khác: sử dụng mẫu save or update, Stream API cho các thao tác truy vấn, defensive copy trong findAll(), và các phương thức kiểm tra tồn tại (existsById()) hay xóa (deleteById()) hoạt động tương tự. Điểm khác biệt nổi bật của repository này là tiền tố ID "SHF", khác với "BK" (Booking), "EMP" (Employee), "INV" (InventoryItem) hay "TXN" (Transaction), phù hợp với entity Shift. Repository cũng cần xử lý ngày tháng bằng LocalDate từ Java Time API, khác với các repository chỉ quản lý dữ liệu cơ bản. Repository cung cấp nhiều phương thức truy vấn hơn, bao gồm findByEmployeeId, findByDate và findByDateRange, hỗ trợ các trường hợp sử dụng như xem lịch làm việc của nhân viên, tạo báo cáo hoặc kiểm tra trùng ca. Phương thức findByDateRange() là query phức tạp nhất, sử dụng logic inclusive range với điều kiện !shift.getDate().isBefore(startDate) và !shift.getDate().isAfter(endDate) để bao gồm cả ngày đầu và ngày cuối, giúp truy vấn ca trong tuần, tháng hoặc bất kỳ khoảng thời gian nào. Ngoài ra, findByDate() còn được sử dụng để hiển thị ca làm việc trong giao diện lịch của ứng dụng, thể hiện sự tích hợp giữa repository layer và application layer. Các hạn chế vẫn tương tự các repository khác: không thread-safe,

thiếu kiểm tra null, không có trừu tượng hóa bằng interface, và logic tạo ID đơn giản (nextId++) không hỗ trợ rollback nếu có lỗi.

3.6 Tầng Model

Tầng Model là lớp dữ liệu nền tảng trong kiến trúc hệ thống, chứa các Domain Entities đại diện cho các nghiệp vụ cốt lõi của nhà hàng.

- Vai trò & Vị trí: Nằm ở tầng thấp nhất, đóng vai trò trung tâm lưu trữ và xử lý trạng thái dữ liệu.
- Đặc điểm kỹ thuật: Các entity không chỉ là vật chứa dữ liệu (Data Containers) mà tích hợp trực tiếp Domain Logic cơ bản (ví dụ: canCancel(), isLowStock()).
- Mục tiêu phân tích: Đi sâu vào từng thành phần mã nguồn để minh chứng quá trình chuyển đổi chính xác từ thiết kế trừu tượng (UML Domain Model) sang hiện thực hóa phần mềm.

3.6.1 Phân tích Booking.java

Booking là phần tử đại diện cho nhiệm vụ đặt bàn, quản lý các thuộc tính cốt lõi gồm thông tin khách hàng, thời gian, bàn được chỉ định và trạng thái hiện tại. Không chỉ dừng lại ở vai trò lưu trữ, lớp này trực tiếp đóng gói logic nghiệp vụ thông qua các phương thức canCancel() và canSeat(), đảm bảo tính hợp lệ của quy trình chuyển đổi trạng thái trước khi tương tác với các thành phần khác như BookingService, BookingController và InMemoryBookingRepository

3.6.1.1 Cấu trúc Class và Khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.model;

import java.time.LocalDate;
import java.time.LocalTime;
```

- **Dòng 1**: Khai báo package com.restaurantmanagement.model xác định vị trí của class trong tầng Domain Entity của kiến trúc phân lớp.
- **Dòng 3-4**: Import LocalDate và LocalTime từ thư viện java.time:

Đoạn mã bắt đầu bằng việc khai báo package com.restaurantmanagement.model, xác định vị trí của lớp Booking nằm trong tầng entity của kiến trúc phân lớp. Các câu lệnh import bao gồm java.time.LocalDate và java.time.LocalTime. Việc sử dụng API thời gian của Java 8 này thay vì lớp java.util.Date cũ mang lại sự an toàn về kiểu dữ liệu (type-safety) và tính bất biến (immutability). Quan trọng hơn, việc tách biệt LocalDate (ngày) và LocalTime (giờ) phản ánh chính xác nghiệp vụ đặt bàn, nơi ngày đặt và giờ đến thường được xử lý độc lập trong các logic tìm kiếm hoặc sắp xếp bàn.

```
/**
* Booking entity representing a restaurant reservation.
*/
public class Booking {
```

- **Dòng 6-8**: Khai báo Class và JavaDoc
- **Dòng 9**: Class Booking được khai báo public, cho phép truy cập tự do từ các tầng Service, Controller và Repository.

Class Booking được khai báo với từ khóa public, cho phép các tầng khác như Service và Repository truy cập tự do. JavaDoc đi kèm mô tả ngắn gọn vai trò của class là đại diện cho một đơn đặt bàn ("restaurant reservation"). Đây là một POJO (Plain Old Java Object) chuẩn, đóng vai trò trung tâm trong việc chuyển giao dữ liệu giữa các lớp của hệ thống

```
    private String id;
```

- **Dòng 10**: Field id (private String):

b) Khai báo Field: ID

Field id được khai báo với kiểu String và access modifier là private. Đây là định danh duy nhất (Primary Key) cho mỗi đơn đặt bàn. Việc sử dụng String thay vì int hay long cho phép linh hoạt trong việc tạo mã định danh tùy chỉnh (ví dụ: "BK-2023001") mà không bị giới hạn bởi kiểu số, đồng thời dễ dàng tương thích với các hệ quản trị cơ sở dữ liệu NoSQL hoặc các API bên ngoài nếu hệ thống mở rộng sau này.

```
        private String customerName;
        private String phoneNumber;
        private int numberOfGuests;
```

- **Dòng 11**: customerName lưu trữ tên khách hàng để định danh người đặt.
- **Dòng 12**: phoneNumber lưu trữ thông tin liên lạc.
- **Dòng 13**: numberOfGuests sử dụng kiểu nguyên thủy int:

c) Khai báo Field: Thông tin Khách hàng

Hai field customerName và phoneNumber lưu trữ thông tin liên hệ thiết yếu của khách hàng. Cả hai đều là kiểu String và private. Trong thiết kế này, thông tin khách hàng được gắn trực tiếp vào Booking thay vì tách ra một bảng Customer riêng biệt (trong phạm vi hiện tại), giúp đơn giản hóa việc truy xuất dữ liệu khi hiển thị danh sách đặt bàn mà không cần thực hiện các phép join phức tạp.

```
        private LocalDate date;
        private LocalTime startTime;
        private String tableId;
        private String status; // CONFIRMED, SEATED, CANCELLED
```

- **Dòng 14**: date (LocalDate) lưu ngày khách đến ăn.
- **Dòng 15**: startTime (LocalTime) lưu giờ bắt đầu.
- **Dòng 16**: tableId lưu tham chiếu đến bàn được gán.
- **Dòng 17**: status lưu trạng thái vòng đời của đơn (CONFIRMED, SEATED, CANCELLED).

d) Khai báo Field: Thời gian và Trạng thái

Các field date, startTime lưu trữ thời điểm khách đến. Field status (kiểu String) quản lý vòng đời của đơn đặt bàn với các giá trị như "CONFIRMED", "SEATED", "CANCELLED". Việc sử dụng String cho trạng thái giúp đơn giản hóa việc lưu trữ và hiển thị, nhưng đòi hỏi sự cẩn trọng khi so sánh chuỗi để tránh lỗi chính tả (typo) trong code logic.

3.6.1.2 Constructors

```
        public Booking() {
        }
```

- **Dòng 19-20**: Constructor rỗng (No-args constructor):

a) Default Constructor

Constructor không tham số public Booking() {} được cung cấp để tuân thủ chuẩn JavaBean. Constructor này rất cần thiết khi sử dụng các framework serialization/deserialization (như Jackson khi làm việc với JSON) hoặc khi khởi tạo đối tượng thông qua Reflection API, cho phép tạo một instance rỗng trước khi gán giá trị qua các phương thức Setter.

```
        public Booking(String id, String customerName, String
phoneNumber, int numberOfGuests,
                       LocalDate date, LocalTime startTime, String
tableId, String status) {
            this.id = id;
            this.customerName = customerName;
            this.phoneNumber = phoneNumber;
            this.numberOfGuests = numberOfGuests;
            this.date = date;
            this.startTime = startTime;
            this.tableId = tableId;
            this.status = status;
        }
```

- **Dòng 22**: Signature nhận đầy đủ 8 tham số.
- **Dòng 24-31**: Thực hiện gán giá trị tham số vào các field tương ứng bằng từ khóa this.

b) Parameterized Constructor

Constructor đầy đủ tham số cho phép khởi tạo một đối tượng Booking với toàn bộ trạng thái hợp lệ ngay từ đầu. Việc sử dụng từ khóa this (ví dụ: this.id = id) giúp phân biệt rõ ràng giữa biến instance và tham số truyền vào. Cách tiếp cận này hỗ trợ tính toàn vẹn dữ liệu, đảm bảo rằng khi một đối tượng được tạo ra thông qua constructor này, nó đã có đầy đủ thông tin cần thiết để sử dụng ngay.

3.6.1.3 Getters và Setters

a) Getter và Setter cho ID

Dòng 35-41: Cặp phương thức truy xuất và gán giá trị cho id. Tuân thủ tính đóng gói, cho phép đọc/ghi giá trị id từ bên ngoài class một cách kiểm soát

b) Getter và Setter cho Customer Name

```
    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }
```

Dòng 43-49: Cho phép truy cập và sửa đổi tên khách hàng. Không chứa logic validation (đây là đặc điểm của Thin Model về mặt dữ liệu).

c) Getter và Setter cho Phone Number

```
    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
```

Dòng 51-57: Tương tự, dùng để quản lý số điện thoại.

d) Getter và Setter cho Number of Guests

```
    public int getNumberOfGuests() {
        return numberOfGuests;
    }

    public void setNumberOfGuests(int numberOfGuests) {
        this.numberOfGuests = numberOfGuests;
    }
```

Dòng 59-65: Làm việc với kiểu dữ liệu nguyên thủy int.

e) Getter và Setter cho Date

```
    public LocalDate getDate() {
        return date;
    }

    public void setDate(LocalDate date) {
        this.date = date;
    }
```

Dòng 67-73: Getter/Setter cho đối tượng LocalDate.

f) Getter và Setter cho Start Time

```
    public LocalTime getStartTime() {
        return startTime;
    }

    public void setStartTime(LocalTime startTime) {
        this.startTime = startTime;
    }
```

Dòng 75-81: Getter/Setter cho đối tượng LocalTime.

g) Getter và Setter cho Table ID

```
    public String getTableId() {
        return tableId;
    }

    public void setTableId(String tableId) {
        this.tableId = tableId;
    }
```

Dòng 83-89: Quản lý liên kết logic tới bàn ăn.

h) Getter và Setter cho Status

```
    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }
```

Dòng 91-97: Quản lý trạng thái. Do status là String, các setter này cần được sử dụng cẩn thận từ Service để đảm bảo tính nhất quán của giá trị (Ví dụ: chỉ set các chuỗi định sẵn như "CONFIRMED").

Hệ thống các phương thức Getter và Setter được khai báo public cho tất cả các field. Các phương thức này tuân thủ tuyệt đối tính đóng gói (Encapsulation), ngăn chặn việc truy cập trực tiếp vào các field private từ bên ngoài. Các Setter ở đây được cài đặt đơn giản, chỉ thực hiện việc gán giá trị (assignment) mà không chứa logic

kiểm tra (validation), phù hợp với nguyên tắc "Thin Model" về mặt dữ liệu, nhường trách nhiệm kiểm tra tính hợp lệ cho tầng Service.

3.6.1.4 Business Logic Methods

a) Method canCancel(): Kiểm tra Booking có thể Hủy

```
        public boolean canCancel() {
            return             !"CANCELLED".equals(status)             &&
!"SEATED".equals(status);
        }
```

- **Dòng 99**: Method trả về boolean, đóng gói domain logic.
- **Dòng 100**: Logic kiểm tra điều kiện hủy.

b) Method canCancel()

Phương thức canCancel() trả về boolean, chứa logic nghiệp vụ quyết định xem một đơn đặt bàn có thể bị hủy hay không. Logic kiểm tra !"CANCELLED".equals(this.status) && !"SEATED".equals(this.status) đảm bảo rằng đơn chỉ được hủy khi nó chưa bị hủy trước đó và khách hàng chưa ngồi vào bàn. Việc đặt logic này trong Entity biến Booking thành một "Rich Domain Model", giúp tập trung quy tắc nghiệp vụ tại nơi chứa dữ liệu, tránh phân tán logic kiểm tra ra khắp nơi trong tầng Service.

c) Method canSeat(): Kiểm tra Booking có thể Chuyển sang SEATED

```
       public boolean canSeat() {
            return "CONFIRMED".equals(status);
        }
```

- **Dòng 106**: Method trả về boolean cho quy trình Check-in khách.
- **Dòng 107**: Logic kiểm tra canSeat():

d) Method canSeat()

Tương tự, phương thức canSeat() kiểm tra điều kiện tiên quyết để xếp bàn cho khách. Logic "CONFIRMED".equals(this.status) quy định chặt chẽ rằng chỉ những đơn đã được xác nhận mới được phép chuyển sang trạng thái "SEATED". Cách viết so sánh chuỗi với hằng số đứng trước (Yoda condition style) giúp tránh lỗi NullPointerException nếu field status vô tình bị null.

e) Method toString()

```
    @Override
    public String toString() {
        return "Booking{" +
                "id='" + id + '\'' +
                ", customerName='" + customerName + '\'' +
                ", date=" + date +
                ", time=" + startTime +
                ", status='" + status + '\'' +
                '}';
```

Dòng 113-121: Override phương thức toString để in ra chuỗi đại diện cho object.

Phương thức toString() được ghi đè (Override) để trả về chuỗi đại diện cho đối tượng dưới dạng JSON-like text. Chuỗi kết quả bao gồm các thông tin quan trọng như ID, tên khách, ngày giờ và trạng thái. Phương thức này cực kỳ hữu ích trong quá trình Debug và Logging, giúp lập trình viên nhanh chóng nắm bắt trạng thái hiện tại của đối tượng mà không cần truy cập từng field riêng lẻ.

3.6.1.5 Tổng kết Booking

Class Booking là một thực thể được thiết kế để cân bằng giữa cấu trúc JavaBean chuẩn mực và logic nghiệp vụ nội tại. Việc sử dụng Java 8 Date/Time API giúp hiện đại hóa việc xử lý thời gian. Đặc biệt, việc tích hợp các phương thức kiểm tra trạng thái (canCancel, canSeat) ngay trong model giúp tăng tính gắn kết (cohesion) của mã nguồn. Tuy nhiên, thuộc tính status đang dùng chuỗi trần (raw string), có thể cải tiến thành Enum để tăng tính an toàn (Type Safety).

3.6.2 Phân tích Employee.java

3.6.2.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
package com.restaurantmanagement.model;
```

Dòng 1: Khai báo package com.restaurantmanagement.model

Class Employee nằm trong package com.restaurantmanagement.model. Điểm đáng chú ý là class này hoàn toàn không có các câu lệnh import. Điều này cho thấy

sự đơn giản của entity: chỉ sử dụng các kiểu dữ liệu nguyên thủy hoặc cơ bản của java.lang (cụ thể là String). Đây là đặc điểm của các entity dạng "Master Data" (dữ liệu danh mục), thường ít biến động và không chứa các logic xử lý phức tạp.

b) Khai báo Class và JavaDoc

```
/**
* Employee entity representing a restaurant employee.
*/
public class Employee {
```

Dòng 3-5: JavaDoc mô tả ngắn gọn vai trò của class: đại diện cho nhân viên nhà hàng. **Dòng 6**: Class được khai báo public, cho phép truy cập từ các tầng Service, Controller và Repository.

c) Khai báo Field

```
private String id;
private String name;
private String position;
private String phoneNumber;
private String email;
```

- **Dòng 7**: Field id (private String):
- **Dòng 8**: name lưu trữ tên nhân viên (thông tin bắt buộc).
- **Dòng 9**: position lưu trữ chức vụ (ví dụ: "Manager", "Chef").
- **Dòng 10**: phoneNumber lưu số điện thoại liên hệ.
- **Dòng 11**: email lưu địa chỉ thư điện tử.

Class bao gồm 5 field private String: id, name, position, phoneNumber, email. Khác với Booking có các ràng buộc trạng thái chặt chẽ, các field của Employee chủ yếu mang tính chất lưu trữ thông tin. Field position lưu trữ chức vụ nhân viên (như "Manager", "Chef"), việc dùng String cho phép nhập liệu linh hoạt nhưng thiếu sự kiểm soát chặt chẽ về danh sách chức vụ cho phép so với việc dùng Enum.

3.6.2.2 Constructors và khởi tạo

a) Default Constructor

```
public Employee() {
}
```

Dòng 13-14: Constructor rỗng tuân thủ chuẩn JavaBean.

b) Parameterized Constructor

```
    public Employee(String id, String name, String position, String
phoneNumber, String email) {
        this.id = id;
        this.name = name;
        this.position = position;
        this.phoneNumber = phoneNumber;
        this.email = email;
    }
```

- **Dòng 16**: Constructor nhận đủ 5 tham số tương ứng với 5 thuộc tính.
- **Dòng 17-21**: Gán giá trị trực tiếp vào các field.

Tương tự Booking, Employee cung cấp cả Default Constructor và Parameterized Constructor. Constructor đầy đủ tham số giúp khởi tạo nhanh các đối tượng nhân viên khi load dữ liệu từ file hoặc database vào bộ nhớ, giảm thiểu số dòng code set giá trị rời rạc.

3.6.2.3 Getters và Setters

a) Getter và Setter cho ID

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

Dòng 25-31: Cặp phương thức truy xuất và gán giá trị cho id.

b) Getter và Setter cho Name

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Dòng 33-39: Cho phép truy cập và sửa đổi tên nhân viên.

c) Getter và Setter cho Position

```
    public String getPosition() {
        return position;
```

```
    }

    public void setPosition(String position) {
        this.position = position;
    }
```

Dòng 41-47: Quản lý thông tin chức vụ.

d) Getter và Setter cho Phone Number

```
    public String getPhoneNumber() {
        return phoneNumber;
    }

    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
```

Dòng 49-55: Quản lý số điện thoại.

e) Getter và Setter cho Email

```
    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
```

Dòng 57-63: Quản lý email.

Các phương thức truy xuất và gán giá trị được cài đặt đầy đủ. Do Employee trong thiết kế này đóng vai trò là "Anemic Domain Model" (Mô hình miền thiếu máu), các setter hoàn toàn không chứa logic nghiệp vụ. Chúng đơn thuần phục vụ việc truyền tải dữ liệu giữa giao diện người dùng (View) và lớp lưu trữ (Repository).

3.6.2.4 Method toString()

```
    @Override
    public String toString() {
        return "Employee{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", position='" + position + '\'' +
                ", phoneNumber='" + phoneNumber + '\'' +
                ", email='" + email + '\'' +
                '}';
```

```
    }
```

- **Dòng 65**: Annotation @Override đánh dấu việc ghi đè phương thức từ class Object.
- **Dòng 66-74**: Trả về chuỗi đại diện chứa **toàn bộ** 5 trường dữ liệu.

Phương thức toString() hiển thị toàn bộ 5 field của nhân viên. Do số lượng thuộc tính ít và tất cả đều là thông tin định danh hoặc liên lạc quan trọng, việc hiển thị đầy đủ giúp ích cho việc log vết hệ thống, đảm bảo khi có lỗi xảy ra, lập trình viên có cái nhìn toàn diện về dữ liệu nhân viên liên quan.

3.6.2.5 Tổng kết Employee

Employee là một entity đơn giản, tuân thủ chặt chẽ chuẩn JavaBean. Nó hoạt động hiệu quả với vai trò là vật chứa dữ liệu (Data Container) cho thông tin nhân sự. Sự vắng mặt của logic nghiệp vụ phức tạp trong class này là hợp lý với ngữ cảnh quản lý thông tin tĩnh, tuy nhiên tính an toàn dữ liệu có thể được nâng cao bằng cách chuẩn hóa field position.

3.6.3 Phân tích InventoryItem.java

InventoryItem là entity đại diện cho hàng tồn kho, quản lý các thông tin về định danh, phân loại, định lượng và vị trí lưu trữ. Khác với Booking hay Employee, entity này nổi bật với khả năng tự động quản lý metadata (thời gian cập nhật) và tích hợp logic kiểm toán tồn kho (isLowStock). Đây là thành phần trung tâm được sử dụng bởi InventoryService để thực hiện các nghiệp vụ nhập/xuất và cảnh báo nguyên vật liệu.

3.6.3.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
    package com.restaurantmanagement.model;

    import java.time.LocalDate;
```

- **Dòng 1**: Khai báo package com.restaurantmanagement.model.
- **Dòng 3**: Import LocalDate từ Java 8 API.

Khác với Booking cần cả giờ phút, quản lý kho hàng (Inventory) thường tập trung vào hạn sử dụng hoặc ngày nhập/cập nhật, do đó LocalDate là lựa chọn chính xác và tối ưu, tránh sự dư thừa thông tin thời gian không cần thiết.

b) Khai báo Class và Field dữ liệu

```
public class InventoryItem {
    private String id;
    private String name;
    private String category; // Food, Beverage, Ingredient, Cleaning,
Others
    private String unit; // kg, liter, piece, box, etc.
    private double quantity;
    private double minimumThreshold;
    private String supplierName;
    private LocalDate lastUpdated;
    private  String  storageLocation;  //  Kitchen,  Bar,  Warehouse,
Freezer
```

- **Dòng 9**: id là khóa chính định danh.
- **Dòng 10-12**: name, category, unit là các thông tin mô tả cơ bản.
- **Dòng 13-14**: quantity và minimumThreshold sử dụng kiểu double để hỗ trợ các đơn vị đo lường thập phân.
- **Dòng 15-17**: Các trường metadata bổ sung.

Định lượng số thực Điểm khác biệt lớn nhất của InventoryItem so với các entity trước là việc sử dụng kiểu dữ liệu double cho quantity và minimumThreshold. Trong nghiệp vụ nhà hàng, nguyên vật liệu không chỉ đếm bằng cái (integer) mà còn đo bằng kg, lít (số thực). Việc sử dụng double phản ánh đúng thực tế nghiệp vụ này.

Field lastUpdated lưu trữ ngày cập nhật cuối cùng của mặt hàng. Đây là thông tin metadata quan trọng để quản lý độ tươi mới của dữ liệu hoặc hỗ trợ quy trình kiểm kê kho.

3.6.3.2 Constructors và quản lý Metadata

a) Default và Parameterized Constructors

```
public InventoryItem() {
    this.lastUpdated = LocalDate.now();
}
```

```
    public InventoryItem(String id, String name, String category, String
unit,
                         double quantity, double minimumThreshold, String
supplierName,
                         String storageLocation) {
        this.id = id;
        this.name = name;
        this.category = category;
        this.unit = unit;
        this.quantity = quantity;
        this.minimumThreshold = minimumThreshold;
        this.supplierName = supplierName;
        this.storageLocation = storageLocation;
        this.lastUpdated = LocalDate.now();
    }
```

Dòng 19-21: Default constructor tự động khởi tạo lastUpdated bằng ngày hiện tại (LocalDate.now()).

Dòng 23-35: Constructor tham số hóa thực hiện logic tương tự cho lastUpdated mà không yêu cầu truyền từ bên ngoài.

b) Cơ chế Automatic Metadata

Trong cả hai constructor (mặc định và có tham số), dòng lệnh this.lastUpdated = LocalDate.now(); được thực thi tự động. Đây là một pattern được thiết kế để thay vì buộc người dùng hoặc service phải nhớ truyền ngày hiện tại vào, entity tự động gán nhãn thời gian ngay khi được khởi tạo. Điều này đảm bảo tính toàn vẹn dữ liệu, ngăn chặn trường hợp item được tạo ra mà không có thông tin ngày tháng (null date).

3.6.3.3 Getters và Setters

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

//… và các getter/setter khác

Dòng 38-108: Hệ thống getter/setter tiêu chuẩn.

3.6.3.4 Business Logic Methods

a) Method isLowStock(): Kiểm tra tồn kho thấp

```
/**
* Check if item is low in stock.
*/
public boolean isLowStock() {
   return quantity <= minimumThreshold;
}
```

Dòng 110-115: Phương thức đóng gói logic nghiệp vụ quan trọng nhất của quản lý kho.

Phương thức isLowStock() trả về boolean, thực hiện so sánh quantity <= minimumThreshold. Đây là logic nghiệp vụ cốt lõi của quản lý kho để kích hoạt cảnh báo nhập hàng. Việc đóng gói logic toán học này vào entity giúp code ở tầng Service trở nên trong sáng hơn (chỉ cần gọi item.isLowStock()) và dễ dàng tái sử dụng ở tầng View để hiển thị cảnh báo đỏ trên giao diện.

b) Method toString()

```
@Override
public String toString() {
   return "InventoryItem{" +
           "id='" + id + '\'' +
           ", name='" + name + '\'' +
           ", category='" + category + '\'' +
           ", quantity=" + quantity +
           ", unit='" + unit + '\'' +
           '}';
}
```

Dòng 117-126: Override toString để hỗ trợ debugging.

Phương thức toString() hiển thị toàn bộ 5 field của nhân viên. Chức năng có tác dụng giống với phương thức toString ở phần Booking

3.6.3.5 Tổng kết InventoryItem

InventoryItem là một ví dụ điển hình về việc mô hình hóa dữ liệu chính xác với thực tế (dùng double cho định lượng). Khả năng tự quản lý metadata (lastUpdated) và tích hợp logic cảnh báo (isLowStock) biến nó thành một thành phần thông minh, giảm tải gánh nặng xử lý cho các tầng khác.

3.6.4 Phân tích InventoryTransaction.java

InventoryTransaction là entity đại diện cho lịch sử giao dịch nhập/xuất kho (audit trail). Khác với InventoryItem phản ánh trạng thái hiện tại, entity này ghi lại dòng chảy dữ liệu (data flow) theo thời gian thực. Điểm nổi bật trong thiết kế của class này là việc sử dụng kỹ thuật **Denormalization** (lưu trữ dư thừa dữ liệu) để đảm bảo tính toàn vẹn của lịch sử giao dịch và **Automatic Timestamping** với độ chính xác cao (LocalDateTime).

3.6.4.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
1  package com.restaurantmanagement.model;
2
3  import java.time.LocalDateTime;
```

- **Dòng 1**: Khai báo package.
- **Dòng 3**: Import LocalDateTime.

Class sử dụng java.time.LocalDateTime. Đối với lịch sử giao dịch (Transaction/Audit Trail), độ chính xác về thời gian là yếu tố sống còn để sắp xếp thứ tự các sự kiện nhập xuất diễn ra trong cùng một ngày. Do đó, LocalDateTime (bao gồm cả ngày và giờ phút giây) là bắt buộc.

b) Khai báo Class và Field dữ liệu

```
5  /**
6  * Inventory Transaction entity representing stock in/out operations.
7  */
8  public class InventoryTransaction {
9     private String id;
10    private String itemId;
11    private String itemName;
12    private double quantity;
13    private String type; // IN or OUT
14    private String reason; // Purchase, Sale, Waste, Adjustment
15    private LocalDateTime timestamp;
16    private String staffId;
17    private String staffName;
```

- **Dòng 9**: id là khóa chính.

- **Dòng 10-11**: itemId và itemName.
- **Dòng 12-14**: quantity sử dụng double. type và reason dùng String
- **Dòng 15**: timestamp lưu thời gian chính xác của giao dịch.
- **Dòng 16-17**: staffId và staffName để lưu thông tin người thực hiện

c) Denormalization Pattern

Class này áp dụng kỹ thuật phi chuẩn hóa (Denormalization). Bên cạnh itemId và staffId (dùng để tham chiếu), class còn lưu trữ thêm itemName và staffName. Mục đích của việc này là tạo ra một bản chụp (Snapshot) dữ liệu tại thời điểm giao dịch. Nếu sau này tên món hàng trong kho bị đổi hoặc nhân viên nghỉ việc và bị xóa khỏi hệ thống, lịch sử giao dịch vẫn hiển thị đúng tên cũ. Điều này cực kỳ quan trọng cho tính trung thực của dữ liệu kiểm toán (Audit Integrity) và tối ưu hóa hiệu năng đọc khi không cần join bảng.

3.6.4.2 Constructors và Automatic Timestamp

Default và Parameterized Constructors

```
      public InventoryTransaction() {
         this.timestamp = LocalDateTime.now();
      }

      public  InventoryTransaction(String  id,  String  itemId,  String
itemName, double quantity,
                                String  type,  String  reason,  String
staffId, String staffName) {
         this.id = id;
         this.itemId = itemId;
         this.itemName = itemName;
         this.quantity = quantity;
         this.type = type;
         this.reason = reason;
         this.staffId = staffId;
         this.staffName = staffName;
         this.timestamp = LocalDateTime.now();
      }
```

Dòng 19-34: Cả hai constructor đều thực hiện logic **Automatic Timestamp Management**.

Tương tự InventoryItem, InventoryTransaction tự động gán this.timestamp = LocalDateTime.now() trong constructor. Điều này đảm bảo rằng thời gian giao dịch

luôn là thời gian thực của hệ thống tại thời điểm ghi nhận, ngăn chặn việc làm giả thời gian giao dịch từ phía client, đảm bảo tính chính xác của nhật ký hoạt động.

3.6.4.3 Getters và Setters

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }
```

// ... các getter/setter khác

Dòng 37-107: Các phương thức truy xuất và gán giá trị tuân thủ chuẩn JavaBean.

3.6.4.4 Method toString()

```
    @Override
    public String toString() {
        return "InventoryTransaction{" +
                "id='" + id + '\'' +
                ", itemName='" + itemName + '\'' +
                ", quantity=" + quantity +
                ", type='" + type + '\'' +
                ", reason='" + reason + '\'' +
                ", timestamp=" + timestamp +
                '}';
    }
```

Dòng 109-119: Override toString phục vụ logging.

Phương thức toString() hiển thị các field của nhân viên. Chức năng có tác dụng giống với phương thức toString ở phần Booking.

3.6.4.5 Tổng kết InventoryTransaction

Entity này được thiết kế chuyên biệt cho mục đích lưu vết. Các quyết định thiết kế như sử dụng LocalDateTime cho độ chính xác cao, tự động gán timestamp, và đặc biệt là lưu trữ dư thừa dữ liệu (tên item, tên staff) cho thấy sự ưu tiên rõ rệt cho tính toàn vẹn của lịch sử và hiệu năng truy xuất báo cáo.

3.6.5 Phân tích Shift.java

Shift là entity đại diện cho phân công ca làm việc, quản lý thông tin về nhân sự, thời gian và loại ca. Entity này tiếp tục áp dụng các pattern đã thấy ở InventoryTransaction như Denormalization, đồng thời sử dụng linh hoạt Java 8 Date/Time API để xử lý tách biệt ngày và giờ, phục vụ cho việc xếp lịch làm việc chi tiết.

3.6.5.1 Cấu trúc Class và khai báo Field

a) Package và Import Statements

```
1  package com.restaurantmanagement.model;
2
3  import java.time.LocalDate;
4  import java.time.LocalTime;
```

Dòng 1: Khai báo package.

Dòng 3-4: Import cả LocalDate và LocalTime.

Package và Time Range Shift import cả LocalDate và LocalTime. Sự khác biệt nằm ở chỗ nó khai báo hai field thời gian: startTime và endTime. Sự kết hợp giữa một ngày cụ thể và một khoảng giờ (Range) giúp định nghĩa chính xác một ca làm việc, làm cơ sở để tính toán tổng giờ công (Duration) sau này.

b) Khai báo Class và Field dữ liệu

```
7  /**
8  * Shift entity representing a work shift assignment.
9  */
10 public class Shift {
11    private String id;
12    private String employeeId;
13    private String employeeName;
14    private LocalDate date;
15    private LocalTime startTime;
16    private LocalTime endTime;
17    private String shiftType; // Morning, Afternoon, Evening, Night
```

- **Dòng 10**: id định danh ca làm việc.
- **Dòng 11-12**: employeeId và employeeName.
- **Dòng 13-15**: date, startTime, endTime.

– **Dòng 16**: shiftType dùng String để phân loại ca.

Khai báo Field: Denormalization

Giống như InventoryTransaction, Shift cũng lưu trữ employeeName bên cạnh employeeId. Việc này phục vụ việc hiển thị lịch làm việc trên giao diện (Calendar View) một cách nhanh chóng mà không cần truy vấn ngược lại thông tin nhân viên, đồng thời giữ cho lịch sử phân công không bị ảnh hưởng nếu dữ liệu nhân viên gốc thay đổi.

3.6.5.2 Constructors và Logic khởi tạo

```
      public Shift() {
      }

      public Shift(String id, String employeeId, String employeeName,
LocalDate date,
                   LocalTime startTime, LocalTime endTime, String
shiftType) {
         this.id = id;
         this.employeeId = employeeId;
         this.employeeName = employeeName;
         this.date = date;
         this.startTime = startTime;
         this.endTime = endTime;
         this.shiftType = shiftType;
      }
```

Dòng 18-30:

Default Constructor: Rỗng, tuân thủ chuẩn JavaBean.

Parameterized Constructor: Gán đầy đủ 7 trường dữ liệu, hỗ trợ khởi tạo nhanh từ dữ liệu form hoặc database.

Khác với các entity kho, Shift không tự động gán ngày giờ hiện tại trong constructor. Lý do nằm ở bản chất nghiệp vụ: Ca làm việc (Shift) là dữ liệu dạng kế hoạch (Plan), thường được tạo cho tương lai. Do đó, các thông tin ngày giờ bắt buộc phải được truyền vào từ bên ngoài (thông qua tham số constructor) thay vì lấy thời gian hệ thống.

3.6.5.3 Getters và Setters

```
      public String getId() {
         return id;
      }
```

```
    public void setId(String id) {
        this.id = id;
    }
```

// ... các getter/setter khác

Dòng 33-87: Các phương thức truy xuất dữ liệu chuẩn.

3.6.5.4 Method toString()

```
    @Override
    public String toString() {
        return "Shift{" +
                "id='" + id + '\'' +
                ", employeeId='" + employeeId + '\'' +
                ", employeeName='" + employeeName + '\'' +
                ", date=" + date +
                ", startTime=" + startTime +
                ", endTime=" + endTime +
                ", shiftType='" + shiftType + '\'' +
                '}';
    }
```

Dòng 89-100: Override toString.

Phương thức toString() hiển thị đầy đủ tất cả các field, bao gồm cả shiftType (loại ca: Sáng/Chiều/Tối). Công dụng của phương thức giống với phương thức toString() của Booking.java.

3.6.5.5 Tổng kết Shift

Shift là entity kết hợp các đặc tính của Booking (quản lý ngày giờ) và InventoryTransaction (lưu trữ dư thừa dữ liệu). Thiết kế của nó hỗ trợ tốt cho việc lập kế hoạch và hiển thị lịch biểu. Việc tách biệt startTime và endTime cung cấp sự linh hoạt để quản lý các ca làm việc có độ dài khác nhau thay vì các khung giờ cố định cứng nhắc.

3.7 Phân tích ứng dụng các nguyên lý OOP trong source code

Các nguyên lý lập trình hướng đối tượng (OOP Object-Oriented Programming) là nền tảng của thiết kế phần mềm hiện đại, giúp tạo ra code dễ bảo trì, mở rộng và tái sử dụng. Dự án Restaurant Management System được xây dựng dựa trên kiến trúc phân lớp (layered architecture) và áp dụng đầy đủ các nguyên lý

OOP cốt lõi: Encapsulation (Đóng gói), Abstraction (Trừu tượng), Inheritance (Kế thừa), và Polymorphism (Đa hình). Phân tích chi tiết này xem xét cách từng nguyên lý được áp dụng trong các tầng khác nhau của hệ thống, với các ví dụ code cụ thể và giải thích mục đích, cách áp dụng và kết quả đạt được.

3.7.1 Encapsulation (Đóng gói)

Encapsulation là nguyên lý đóng gói dữ liệu và các phương thức liên quan vào trong một class, đồng thời kiểm soát quyền truy cập vào dữ liệu đó thông qua các access modifiers (private, protected, public). Nguyên lý này đảm bảo tính toàn vẹn dữ liệu, giảm thiểu sự phụ thuộc giữa các class, và tạo điều kiện cho việc bảo trì code.

3.7.1.1 Encapsulation trong tầng Model

a) Mục đích

Trong tầng Model, encapsulation được sử dụng để:

- Bảo vệ dữ liệu của các entity khỏi việc truy cập trực tiếp từ bên ngoài
- Đảm bảo tính toàn vẹn dữ liệu thông qua các getter/setter methods
- Tuân theo JavaBean convention để tương thích với các framework như JavaFX

b) Cách áp dụng

Tất cả các fields trong các entity classes đều được khai báo là 'private', và chỉ có thể truy cập thông qua các public getter/setter methods.

Ví dụ trong 'Booking.java'

```
private String id;
private String customerName;
private String phoneNumber;
private int numberOfGuests;
private LocalDate date;
private LocalTime startTime;
private String tableId;
private String status; // CONFIRMED, SEATED, CANCELLED
```

Các getter/setter methods

```
public String getId() {
    return id;
}

public void setId(String id) {
```

```
    this.id = id;
}

public String getCustomerName() {
    return customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}

// ... các getter/setter khác
```

Tương tự trong các entity khác:

- 'Employee.java': Tất cả 5 fields ('id', 'name', 'position', 'phoneNumber', 'email') đều là 'private' với getter/setter tương ứng
- 'InventoryItem.java': Tất cả 9 fields đều là 'private' với getter/setter tương ứng
- 'Shift.java': Tất cả 7 fields đều là 'private' với getter/setter tương ứng
- 'InventoryTransaction.java': Tất cả 9 fields đều là 'private' với getter/setter tương ứng

c) Kết quả

- **Bảo vệ dữ liệu**: Không thể truy cập trực tiếp vào fields từ bên ngoài class, đảm bảo dữ liệu không bị thay đổi một cách không kiểm soát
- **Tính nhất quán**: Tất cả các entity đều tuân theo cùng một pattern (JavaBean convention), dễ dàng sử dụng với JavaFX Property binding
- **Khả năng mở rộng**: Có thể thêm validation logic vào setters trong tương lai mà không cần thay đổi code sử dụng entity
- **Tương thích framework**: JavaFX có thể tự động bind properties thông qua naming convention (getXxx/setXxx)

3.7.1.2 Encapsulation trong Tầng Service

a) Mục đích

Trong tầng Service, encapsulation được sử dụng để:

- Ẩn implementation details của repository khỏi controller

- Bảo vệ business logic và validation methods khỏi truy cập trực tiếp
- Đảm bảo chỉ các operations hợp lệ mới được thực hiện

b) Cách áp dụng

Repository dependency được đóng gói:

```
public class BookingService {
    private final InMemoryBookingRepository repository;
```

Private validation methods:

```
/**
 * Validate booking data.
 */
private void validateBooking(Booking booking) {
    if (booking == null) {
        throw new IllegalArgumentException("Booking cannot be null");
    }
    if (booking.getCustomerName() == null ||
booking.getCustomerName().trim().isEmpty()) {
        throw new IllegalArgumentException("Customer name is
required");
    }
    // ... các validation khác
}
```

Tương tự trong các service khác:

- 'EmployeeService': Có 'private validateEmployee()' method
- 'InventoryService': Có 'private validateItem()' method
- 'ShiftService': Có 'private validateShift()' method

c) Kết quả

- **Ẩn implementation:** Controller không cần biết service sử dụng repository nào, chỉ cần gọi public methods
- **Bảo vệ business logic:** Validation methods là 'private', không thể bị bỏ qua từ bên ngoài
- **Tính nhất quán:** Tất cả operations đều phải đi qua validation trước khi thực hiện
- **Dễ bảo trì:** Có thể thay đổi implementation của repository mà không ảnh hưởng đến controller

3.7.1.3 Encapsulation trong Tầng Repository

a) Mục đích

Trong tầng Repository, encapsulation được sử dụng để:

- Bảo vệ cấu trúc dữ liệu nội bộ (ArrayList) khỏi truy cập trực tiếp
- Đảm bảo ID generation logic được kiểm soát
- Ngăn chặn việc thao tác dữ liệu không đúng cách

b) Cách áp dụng

Internal data structure được đóng gói:

```
public class InMemoryBookingRepository {
    private final List<Booking> bookings;
    private int nextId = 1;
}
```

Defensive copy trong 'findAll()':

```
/**
 * Get all bookings.
 */
public List<Booking> findAll() {
    return new ArrayList<>(bookings);
}
```

ID generation logic được đóng gói trong 'save()':

```
/**
 * Save a new booking or update an existing one.
 */
public Booking save(Booking booking) {
    if (booking.getId() == null || booking.getId().isEmpty()) {
        // New booking assign ID
        booking.setId("BK" + String.format("%04d", nextId++));
        bookings.add(booking);
        return booking;
    } else {
        // Update existing booking
        Optional<Booking> existing = findById(booking.getId());
        if (existing.isPresent()) {
            int index = bookings.indexOf(existing.get());
            bookings.set(index, booking);
            return booking;
        } else {
            bookings.add(booking);
            return booking;
        }
    }
}
```

c) Kết quả

- **Bảo vệ dữ liệu**: List nội bộ không thể bị thay đổi trực tiếp từ bên ngoài
- **Tính toàn vẹn**: ID generation được kiểm soát, đảm bảo không có ID trùng lặp

- **Defensive programming**: ‘findAll()’ trả về defensive copy, ngăn chặn việc thay đổi list gốc từ bên ngoài
- **Dễ thay đổi implementation**: Có thể thay đổi từ ArrayList sang cấu trúc dữ liệu khác mà không ảnh hưởng đến service layer

3.7.1.4 Encapsulation trong Tầng Controller

a) Mục đích

Trong tầng Controller, encapsulation được sử dụng để:

- Bảo vệ service dependencies và UI component references
- Quản lý ObservableList nội bộ
- Ẩn implementation details của UI event handling

b) Cách áp dụng

Service dependency và ObservableList được đóng gói:

```
public class BookingController {
    private final BookingService bookingService;
    private final ObservableList<Booking> bookingList;
    private TableView<Booking> tableView;
```

UI component references được đóng gói:

```
// UI Components (for form inputs)
private TextField idField;
private TextField customerNameField;
private TextField phoneNumberField;
private TextField numberOfGuestsField;
private DatePicker datePicker;
private ComboBox<Integer> hourComboBox;
private ComboBox<Integer> minuteComboBox;
private TextField tableIdField;
private ComboBox<String> statusComboBox;
```

Private helper methods:

```
private void showSuccessAlert(String message) {
    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle("Success");
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
}

private void showErrorAlert(String message) {
    Alert alert = new Alert(Alert.AlertType.ERROR);
    alert.setTitle("Error");
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
}
```

c) Kết quả

- **Bảo vệ dependencies**: Service và UI components không thể bị thay đổi trực tiếp từ bên ngoài
- **Quản lý state**: ObservableList được quản lý nội bộ, đảm bảo tính nhất quán với UI
- **Tái sử dụng code**: Helper methods như 'showSuccessAlert()' và 'showErrorAlert()' được sử dụng nhiều lần trong class
- **Dễ test**: Có thể inject mock service thông qua constructor để test

3.7.1.5 Encapsulation trong Tầng App (MainApp)

a) Mục đích

Trong tầng App, encapsulation được sử dụng để:

- Bảo vệ controller instances và UI component references
- Quản lý state của application (như 'currentWeekStart')
- Ẩn implementation details của UI creation

b) Cách áp dụng

Controller instances và UI components được đóng gói:

```
public class MainApp extends Application {
    private EmployeeController employeeController;
    private ShiftController shiftController;
    private InventoryController inventoryController;
    private BookingController bookingController;
    private TableView<Employee> employeeTable;
    private TableView<Shift> shiftTable;
    private TableView<InventoryItem> inventoryTable;
    private TableView<InventoryTransaction> transactionTable;
    private TableView<Booking> bookingTable;
    private TextField idField, nameField, positionField, phoneField,
emailField;
    private TextField shiftIdField;
    private ComboBox<String> employeeComboBox;
    private DatePicker datePicker;
    private ComboBox<Integer> startHourComboBox, startMinuteComboBox;
    private ComboBox<Integer> endHourComboBox, endMinuteComboBox;
    private ComboBox<String> shiftTypeComboBox;
    private LocalDate currentWeekStart;
}
```

Private helper methods cho UI creation:

```
/**
 * Create the main application layout with navigation tabs.
 */
private VBox createMainLayout() {
    VBox root = new VBox();
    root.setStyle("-fx-background-color: #f5f5f5;");

    // Title Bar
    HBox titleBar = createTitleBar();

    // Tab Navigation
    TabPane tabPane = createTabPane();
    tabPane.setTabClosingPolicy(TabPane.TabClosingPolicy.UNAVAILABLE)
;

    root.getChildren().addAll(titleBar, tabPane);
    VBox.setVgrow(tabPane, Priority.ALWAYS);

    return root;
}
```

c) Kết quả

- **Bảo vệ state**: Application state được quản lý nội bộ, không thể bị thay đổi từ bên ngoài
- **Tổ chức code**: UI creation logic được tách thành các private methods, code dễ đọc và bảo trì hơn
- **Tái sử dụng**: Các helper methods có thể được gọi nhiều lần trong class
- **Dễ mở rộng**: Có thể thêm UI components mới mà không ảnh hưởng đến các phần khác

3.7.1.6 Tổng kết Encapsulation

- **Encapsulation được áp dụng nhất quán trong toàn bộ hệ thống:**
  - **Model Layer**: Tất cả fields đều là 'private' với public getter/setter, tuân theo JavaBean convention
  - **Service Layer**: Repository dependencies và validation methods là 'private', chỉ expose public business methods
  - **Repository Layer**: Internal data structures và ID generation logic được bảo vệ, trả về defensive copies
  - **Controller Layer**: Service dependencies, ObservableList, và UI components được đóng gói, helper methods là 'private'

- **App Layer**: Controller instances, UI components, và application state được quản lý nội bộ

– **Lợi ích tổng thể:**
  - **Bảo mật dữ liệu:** Dữ liệu không thể bị truy cập hoặc thay đổi trực tiếp
  - **Tính nhất quán:** Tất cả các tầng đều tuân theo cùng một pattern encapsulation
  - **Dễ bảo trì:** Có thể thay đổi implementation mà không ảnh hưởng đến các tầng khác
  - **Dễ test:** Có thể inject dependencies thông qua constructor để test

3.7.2 Abstraction (Trừu tượng)

Abstraction là nguyên lý ẩn đi các chi tiết implementation phức tạp và chỉ expose những gì cần thiết cho người sử dụng. Trong dự án này, abstraction được thể hiện qua Repository Pattern, Service Layer abstraction, và method abstraction trong các classes.

3.7.2.1 Repository Pattern Abstraction của Data Access Layer

a) Mục đích

Repository Pattern được sử dụng để:

– Ẩn đi chi tiết implementation của data storage (hiện tại là in-memory, có thể thay đổi thành database)
– Cung cấp interface thống nhất cho các thao tác CRUD
– Cho phép thay đổi persistence mechanism mà không ảnh hưởng đến service layer

b) Cách áp dụng

Abstraction qua consistent interface:

Tất cả các repository classes đều có cùng một set methods, mặc dù implementation có thể khác nhau:

```
public Booking save(Booking booking) {
```

```
    if (booking.getId() == null || booking.getId().isEmpty()) {
        // New booking assign ID
        booking.setId("BK" + String.format("%04d", nextId++));
        bookings.add(booking);
        return booking;
    } else {
        // Update existing booking
        Optional<Booking> existing = findById(booking.getId());
        if (existing.isPresent()) {
            int index = bookings.indexOf(existing.get());
            bookings.set(index, booking);
            return booking;
        } else {
            bookings.add(booking);
            return booking;
        }
    }
}
```

Service layer sử dụng abstraction:

```
public class BookingService {
    private final InMemoryBookingRepository repository;

    public BookingService() {
        this.repository = new InMemoryBookingRepository();
    }
```

Service chỉ biết về interface của repository (các methods như 'save()', 'findById()', 'findAll()'), không cần biết chi tiết implementation (ArrayList, ID generation logic).

- **Tương tự trong các repository khác:**
    - 'InMemoryEmployeeRepository': Cùng pattern với prefix "EMP"
    - 'InMemoryInventoryRepository': Cùng pattern với prefix "INV"
    - 'InMemoryShiftRepository': Cùng pattern với prefix "SHF"
    - 'InMemoryInventoryTransactionRepository': Cùng pattern với prefix "TXN"

c) Kết quả

- **Tách biệt concerns**: Service layer không phụ thuộc vào chi tiết implementation của data storage
- **Dễ thay đổi**: Có thể thay thế 'InMemoryBookingRepository' bằng 'DatabaseBookingRepository' mà không cần sửa 'BookingService'

- **Tính nhất quán**: Tất cả repositories đều có cùng interface, dễ dàng hiểu và sử dụng
- **Dễ test**: Có thể tạo mock repository để test service layer
- **Lưu ý:** Trong code hiện tại, service sử dụng implementation cụ thể ('InMemoryBookingRepository') thay vì interface. Điều này có thể được cải thiện bằng cách tạo interface 'BookingRepository' và để 'InMemoryBookingRepository' implement interface đó.

3.7.2.2 Service Layer Abstraction

a) Mục đích

Service layer cung cấp abstraction cho:

- Business logic operations
- Data validation
- Transaction coordination
- Ẩn đi chi tiết của repository operations

b) Cách áp dụng

Abstraction qua business methods:

```
/**
 * Add a new booking.
 */
public Booking addBooking(Booking booking) {
    validateBooking(booking);
    return repository.save(booking);
}
```

Controller chỉ cần gọi 'addBooking()', không cần biết về validation logic hoặc cách repository lưu dữ liệu.

Complex business operations:

```
/**
 * Cancel a booking.
 */
public Booking cancelBooking(String id) {
    Optional<Booking> bookingOpt = repository.findById(id);
    if (!bookingOpt.isPresent()) {
        throw new IllegalArgumentException("Booking with ID " + id +
" not found");
    }

    Booking booking = bookingOpt.get();
```

```
    if (!booking.canCancel()) {
        throw new IllegalArgumentException("Booking cannot be
cancelled. Current status: " + booking.getStatus());
    }

    booking.setStatus("CANCELLED");
    return repository.save(booking);
}
```

Controller chỉ cần gọi ‘cancelBooking(id)’, không cần biết về business rules (kiểm tra status, validation, v.v.).

InventoryService với complex operations:

```
/**
 * Stock In: Add quantity to an item.
 */
public InventoryTransaction stockIn(String itemId, double quantity,
String reason,
                                    String staffId, String staffName)
{
    if (quantity <= 0) {
        throw new IllegalArgumentException("Quantity must be greater
than 0");
    }

    Optional<InventoryItem> itemOpt =
itemRepository.findById(itemId);
    if (!itemOpt.isPresent()) {
        throw new IllegalArgumentException("Item with ID " + itemId +
" not found");
    }

    InventoryItem item = itemOpt.get();
    item.setQuantity(item.getQuantity() + quantity);
    item.setLastUpdated(LocalDate.now());
    itemRepository.save(item);

    // Create transaction record
    InventoryTransaction transaction = new InventoryTransaction();
    transaction.setItemId(itemId);
    transaction.setItemName(item.getName());
    transaction.setQuantity(quantity);
    transaction.setType("IN");
    transaction.setReason(reason);
    transaction.setStaffId(staffId);
    transaction.setStaffName(staffName);

    return transactionRepository.save(transaction);
}
```

Controller chỉ cần gọi ‘stockIn()’ với các tham số, không cần biết về logic cập nhật quantity, tạo transaction record, v.v.

c) Kết quả

- **Đơn giản hóa controller**: Controller không cần biết về business logic phức tạp
- **Tái sử dụng**: Business logic có thể được sử dụng bởi nhiều controllers hoặc từ nhiều nơi khác nhau
- **Dễ test**: Có thể test business logic độc lập với UI
- **Dễ bảo trì**: Business rules được tập trung ở một nơi, dễ dàng cập nhật

3.7.2.3 Method Abstraction trong Classes

a) Mục đích

Method abstraction được sử dụng để:

- Ẩn đi implementation details của các operations phức tạp
- Tạo các helper methods để tái sử dụng code
- Tổ chức code thành các methods nhỏ, dễ đọc

b) Cách áp dụng

Abstraction trong MainApp UI creation methods:

```
private VBox createCalendarView() {
    // Complex calendar view creation logic
    // ...
}
```

Method này ẩn đi toàn bộ logic phức tạp của việc tạo calendar view, chỉ expose method name và return type.

Abstraction trong Controller Helper methods:

```
private void showSuccessAlert(String message) {
    Alert alert = new Alert(Alert.AlertType.INFORMATION);
    alert.setTitle("Success");
    alert.setHeaderText(null);
    alert.setContentText(message);
    alert.showAndWait();
}
```

Method này abstract away việc tạo và hiển thị alert, chỉ cần truyền message.

Abstraction trong Service Validation methods:

```
/**
 * Validate booking data.
 */
private void validateBooking(Booking booking) {
    if (booking == null) {
        throw new IllegalArgumentException("Booking cannot be null");
    }
```

```
    if (booking.getCustomerName() == null ||
booking.getCustomerName().trim().isEmpty()) {
        throw new IllegalArgumentException("Customer name is
required");
    }
    // ... các validation khác
}
```

Method này abstract away validation logic, các methods khác chỉ cần gọi 'validateBooking()'.

c) Kết quả

- **Code dễ đọc**: Methods có tên mô tả rõ ràng, dễ hiểu mục đích
- **Tái sử dụng**: Helper methods có thể được gọi nhiều lần
- **Dễ bảo trì**: Logic phức tạp được tập trung ở một nơi, dễ dàng sửa đổi
- **Giảm duplication**: Tránh lặp lại code bằng cách tạo helper methods

3.7.2.4 Tổng kết Abstraction

- **Abstraction được áp dụng ở nhiều mức độ trong hệ thống:**
  - **Architectural Level**: Repository Pattern ẩn đi data access implementation
  - **Service Level**: Service layer ẩn đi business logic complexity
  - **Method Level**: Helper methods ẩn đi implementation details
- **Lợi ích tổng thể:**
  - **Giảm complexity**: Người sử dụng chỉ cần biết interface, không cần biết implementation
  - **Tăng tính linh hoạt**: Có thể thay đổi implementation mà không ảnh hưởng đến code sử dụng
  - **Dễ bảo trì**: Logic phức tạp được tập trung, dễ dàng cập nhật
  - **Tái sử dụng**: Code được tổ chức tốt, dễ dàng tái sử dụng

3.7.3 Inheritance (Kế thừa)

Inheritance là nguyên lý cho phép một class kế thừa các thuộc tính và phương thức từ một class khác. Trong dự án này, inheritance được sử dụng ở mức độ hạn chế, chủ yếu là kế thừa từ các class của Java và JavaFX framework.

3.7.3.1 Inheritance từ Object Class

a) Mục đích

Tất cả các class trong Java đều kế thừa từ 'Object' class, cho phép:

- Sử dụng các methods cơ bản như 'toString()', 'equals()', 'hashCode()'
- Override các methods này để cung cấp implementation phù hợp với từng class

b) Cách áp dụng

Override 'toString()' method:

Tất cả các entity classes đều override 'toString()' method từ 'Object' class:

```
@Override
public String toString() {
    return "Booking{" +
            "id='" + id + '\'' +
            ", customerName='" + customerName + '\'' +
            ", date=" + date +
            ", time=" + startTime +
            ", status='" + status + '\'' +
            '}';
}
```

- **Tương tự trong các entity khác:**
  - 'Employee.toString()': Override để trả về string representation của employee
  - 'InventoryItem.toString()': Override để trả về string representation của inventory item
  - 'Shift.toString()': Override để trả về string representation của shift
  - 'InventoryTransaction.toString()': Override để trả về string representation của transaction
- **Annotation '@Override':**
  - Annotation '@Override' đánh dấu method này override method từ parent class
  - Compiler sẽ kiểm tra method signature có khớp với method trong parent class không
  - Giúp tránh lỗi typo hoặc signature không khớp

c) Kết quả

- **Consistent interface**: Tất cả objects đều có 'toString()' method, có thể in ra console hoặc log
- **Debugging support**: Dễ dàng debug bằng cách in object ra console
- **Polymorphism**: Có thể gọi 'toString()' trên bất kỳ object nào mà không cần biết kiểu cụ thể
- **Best practice**: Override 'toString()' là best practice trong Java

3.7.3.2 Inheritance từ Application Class (JavaFX)

a) Mục đích

'MainApp' class kế thừa từ 'Application' class của JavaFX để:

- Tạo JavaFX application entry point
- Override 'start()' method để khởi tạo UI
- Tích hợp với JavaFX application lifecycle

b) Cách áp dụng

Class declaration:

```
public class MainApp extends Application {
```

Override 'start()' method:

```
@Override
public void start(Stage primaryStage) {
    // Create shared EmployeeService instance so both controllers use
the same data
    EmployeeService sharedEmployeeService = new EmployeeService();
    employeeController = new
EmployeeController(sharedEmployeeService);
    shiftController = new ShiftController(sharedEmployeeService);
    inventoryController = new InventoryController();
    bookingController = new BookingController();

    // Initialize current week to start of current week
    currentWeekStart =
LocalDate.now().with(java.time.DayOfWeek.MONDAY);

    primaryStage.setTitle("Restaurant Management System");
    primaryStage.setWidth(1600);
    primaryStage.setHeight(900);
    primaryStage.setMinWidth(1200);
    primaryStage.setMinHeight(700);

    VBox root = createMainLayout();
    Scene scene = new Scene(root);
    primaryStage.setScene(scene);
```

```
    primaryStage.show();
}
```

Inherited methods và properties:

- 'MainApp' kế thừa tất cả methods từ 'Application' class
- Có thể sử dụng 'launch()' method để start application
- Có thể override các lifecycle methods như 'init()', 'stop()' nếu cần

c) Kết quả

- **Framework integration:** Tích hợp với JavaFX framework, có thể sử dụng JavaFX features
- **Application lifecycle:** Quản lý application lifecycle (start, stop, init)
- **Code reuse:** Kế thừa functionality từ 'Application' class, không cần implement lại
- **Standard pattern:** Tuân theo standard pattern của JavaFX applications

3.7.3.3 Không có Abstract Classes hoặc Interfaces trong Domain Model

a) Quan sát

Trong codebase hiện tại, không có abstract classes hoặc interfaces được định nghĩa trong domain model. Tất cả các classes đều là concrete classes.

- **Lý do có thể:**
  - Domain model đơn giản, không cần abstraction ở mức này
  - Các entities không có nhiều điểm chung để tạo base class
  - Repository pattern chưa sử dụng interface (có thể được cải thiện)
- **Cơ hội cải thiện:**
  - Có thể tạo interface 'Repository<T>' để abstract repository operations
  - Có thể tạo abstract base class cho các entities nếu có nhiều common functionality
  - Có thể tạo interface 'Service' nếu cần multiple implementations

b) Kết quả

- **Đơn giản**: Code đơn giản, dễ hiểu, không có nhiều layers của inheritance
- **Flexibility**: Có thể thêm abstraction trong tương lai nếu cần

- **Trade-off**: Không có polymorphism qua inheritance trong domain model, nhưng có thể sử dụng composition thay thế

3.7.3.4 Tổng kết Inheritance

- Inheritance được sử dụng ở mức độ hạn chế trong dự án:
  - **Implicit inheritance**: Tất cả classes đều kế thừa từ 'Object' class
  - **Framework inheritance**: 'MainApp' kế thừa từ 'Application' class của JavaFX
  - **Method overriding**: Override 'toString()' method trong tất cả entity classes
- **Lợi ích:**
  - **Code reuse:** Kế thừa functionality từ parent classes
  - **Framework integration:** Tích hợp với JavaFX framework
  - **Consistent interface:** Override 'toString()' để có consistent interface
- **Hạn chế:**
  - Không có domain-specific inheritance hierarchy
  - Có thể được cải thiện bằng cách tạo interfaces cho repositories

3.7.4 Polymorphism (Đa hình)

Polymorphism là nguyên lý cho phép các objects của các class khác nhau được xử lý thông qua một interface chung. Trong dự án này, polymorphism được thể hiện qua method overriding, method overloading, và runtime polymorphism với JavaFX components.

3.7.4.1 Method Overriding (Runtime Polymorphism)

a) Mục đích

Method overriding cho phép:

- Cung cấp implementation cụ thể cho methods được kế thừa
- Thay đổi behavior của methods từ parent class

– Đạt được runtime polymorphism method được gọi được quyết định tại runtime dựa trên object type

b) Cách áp dụng

Override 'toString()' trong các entity classes:

Mỗi entity class override 'toString()' với implementation khác nhau:

```
@Override
public String toString() {
    return "Booking{" +
            "id='" + id + '\'' +
            ", customerName='" + customerName + '\'' +
            ", date=" + date +
            ", time=" + startTime +
            ", status='" + status + '\'' +
            '}';
}
```

```
@Override
public String toString() {
    return "Employee{" +
            "id='" + id + '\'' +
            ", name='" + name + '\'' +
            ", position='" + position + '\'' +
            ", phoneNumber='" + phoneNumber + '\'' +
            ", email='" + email + '\'' +
            '}';
}
```


Override 'start()' trong MainApp:

```
@Override
public void start(Stage primaryStage) {
    // Custom implementation for JavaFX application startup
    // ...
}
```

c) Kết quả

– Runtime polymorphism: Khi gọi 'object.toString()', method được gọi phụ thuộc vào actual type của object tại runtime

– Custom behavior: Mỗi class có thể cung cấp implementation phù hợp với nhu cầu của mình

– Consistent interface: Tất cả objects đều có 'toString()' method, nhưng behavior khác nhau

– Framework integration: Override 'start()' để tích hợp với JavaFX lifecycle

Ví dụ runtime polymorphism:

```
Object obj1 = new Booking();
Object obj2 = new Employee();
System.out.println(obj1.toString()); // Calls Booking.toString()
System.out.println(obj2.toString()); // Calls Employee.toString()
```

3.7.4.2 Method Overloading (Compile-time Polymorphism)

a) Mục đích

Method overloading cho phép:

- Định nghĩa nhiều methods với cùng tên nhưng khác signature (parameters)
- Cung cấp nhiều cách gọi method với các kiểu dữ liệu khác nhau
- Tăng tính linh hoạt và dễ sử dụng của API

b) Cách áp dụng

Constructor overloading trong entity classes:

Tất cả entity classes đều có ít nhất 2 constructors:

```
public Booking() {
}
public Booking(String id, String customerName, String phoneNumber,
int numberOfGuests,
               LocalDate date, LocalTime startTime, String tableId,
String status) {
    this.id = id;
    this.customerName = customerName;
    this.phoneNumber = phoneNumber;
    this.numberOfGuests = numberOfGuests;
    this.date = date;
    this.startTime = startTime;
    this.tableId = tableId;
    this.status = status;
}
```

- Tương tự trong các entity khác:
  - 'Employee': Default constructor và parameterized constructor
  - 'InventoryItem': Default constructor và parameterized constructor
  - 'Shift': Default constructor và parameterized constructor
  - 'InventoryTransaction': Default constructor và parameterized constructor

Controller constructor overloading:

```
public BookingController() {
    this.bookingService = new BookingService();
    this.bookingList = FXCollections.observableArrayList();
    loadBookings();
}
```

Một số controllers có thêm constructor với dependency injection:

```
public EmployeeController() {
    this.employeeService = new EmployeeService();
    this.employeeList = FXCollections.observableArrayList();
    loadEmployees();
}

public EmployeeController(EmployeeService employeeService) {
    this.employeeService = employeeService;
    this.employeeList = FXCollections.observableArrayList();
    loadEmployees();
```

c) Kết quả

- **Flexibility**: Có thể tạo object bằng nhiều cách khác nhau
- **Convenience**: Default constructor cho phép tạo object rồi set properties sau
- **Dependency injection**: Constructor với parameters cho phép inject dependencies
- **Backward compatibility**: Có thể thêm constructors mới mà không ảnh hưởng đến code cũ

3.7.4.3 Runtime Polymorphism với JavaFX Components

a) Mục đích

JavaFX sử dụng polymorphism để:

- Xử lý các UI components thông qua base classes và interfaces
- Cho phép thay đổi behavior tại runtime
- Hỗ trợ event handling với lambda expressions

b) Cách áp dụng

Polymorphism với JavaFX Control hierarchy:

```
private TableView<Employee> employeeTable;
private TableView<Shift> shiftTable;
private TableView<InventoryItem> inventoryTable;
private TableView<InventoryTransaction> transactionTable;
private TableView<Booking> bookingTable;
```

Tất cả đều là ‘TableView<T>‘, nhưng với generic type khác nhau, thể hiện polymorphism qua generics.

Polymorphism với event handlers:

```
public void handleAdd() {
    try {
        Booking booking = createBookingFromForm();
        bookingService.addBooking(booking);
        loadBookings();
        clearForm();
        showSuccessAlert("Booking added successfully!");
    } catch (Exception e) {
        showErrorAlert("Error adding booking: " + e.getMessage());
    }
}
```

Method ‘handleAdd()’ có thể được gọi từ nhiều nơi khác nhau (button click, menu item, v.v.), thể hiện polymorphism.

Polymorphism với ObservableList:

```
private final ObservableList<Booking> bookingList;
```

‘ObservableList’ là interface, có thể sử dụng với nhiều implementations khác nhau, thể hiện polymorphism qua interface.

c) Kết quả

- **Flexibility**: Có thể thay đổi UI components mà không cần thay đổi code xử lý
- **Code reuse**: Có thể xử lý nhiều loại components thông qua cùng một interface
- **Type safety**: Generics đảm bảo type safety trong khi vẫn có polymorphism
- **Event handling**: Lambda expressions cho phép flexible event handling

3.7.4.4 Polymorphism qua Stream API

a) Mục đích

Stream API sử dụng polymorphism để:

- Xử lý collections thông qua functional interfaces
- Cho phép method references và lambda expressions
- Hỗ trợ functional programming patterns

b) Cách áp dụng

Method references:

```
public List<InventoryItem> findLowStockItems() {
```

```
    return items.stream()
            .filter(InventoryItem::isLowStock)
            .collect(Collectors.toList());
}
```

'InventoryItem::isLowStock' là method reference, thể hiện polymorphism có thể truyền method như một parameter.

Lambda expressions:

```
public Optional<Booking> findById(String id) {
    return bookings.stream()
            .filter(booking -> booking.getId().equals(id))
            .findFirst();
}
```

Lambda expression 'booking -> booking.getId().equals(id)' là implementation của functional interface 'Predicate<Booking>', thể hiện polymorphism.

Polymorphism với functional interfaces:

```
public List<Booking> findByDate(java.time.LocalDate date) {
    return bookings.stream()
            .filter(booking -> booking.getDate() != null &&
booking.getDate().equals(date))
            .collect(Collectors.toList());
}
```

'filter()' method nhận 'Predicate<T>' interface, có thể nhận lambda expression hoặc method reference, thể hiện polymorphism.

c) Kết quả

- **Functional programming**: Hỗ trợ functional programming patterns với lambda và method references
- **Code conciseness**: Code ngắn gọn và dễ đọc hơn
- **Flexibility**: Có thể truyền behavior như parameters
- **Type safety**: Functional interfaces đảm bảo type safety

3.7.4.5 Tổng kết Polymorphism

- **Polymorphism được áp dụng ở nhiều mức độ trong hệ thống:**
    - **Method Overriding**: Override 'toString()' và 'start()' để cung cấp custom behavior
    - **Method Overloading**: Constructor overloading để tạo objects bằng nhiều cách
    - **Runtime Polymorphism**: JavaFX components và event handling

    - **Functional Polymorphism**: Stream API với lambda expressions và method references
- **Lợi ích tổng thể**:
    - **Flexibility**: Code linh hoạt, có thể thay đổi behavior tại runtime
    - **Code reuse**: Có thể xử lý nhiều types thông qua cùng một interface
    - **Maintainability**: Dễ dàng thay đổi implementation mà không ảnh hưởng đến code sử dụng
    - **Expressiveness**: Code dễ đọc và biểu đạt hơn với lambda và method references

3.7.5 Tổng kết và đánh giá

3.7.5.1 Tổng hợp các nguyên lý OOP

Dự án Restaurant Management System đã áp dụng đầy đủ 4 nguyên lý OOP cốt lõi:

- **Encapsulation**: Được áp dụng nhất quán trong tất cả các tầng, bảo vệ dữ liệu và implementation details
- **Abstraction**: Thể hiện qua Repository Pattern, Service Layer, và method abstraction
- **Inheritance**: Sử dụng ở mức độ hạn chế, chủ yếu kế thừa từ 'Object' và 'Application' class
- **Polymorphism**: Được áp dụng qua method overriding, overloading, và runtime polymorphism với JavaFX

3.7.5.2 Điểm mạnh

- **Consistent patterns**: Tất cả các tầng đều tuân theo cùng một pattern, dễ hiểu và bảo trì
- **Good encapsulation**: Dữ liệu được bảo vệ tốt, chỉ truy cập qua public methods
- **Clear abstraction**: Repository Pattern và Service Layer tạo abstraction rõ ràng
- **Functional programming support**: Sử dụng Stream API với lambda và method references

3.7.5.3 Cơ hội cải thiện

- **Interface-based design**: Có thể tạo interfaces cho repositories để tăng tính linh hoạt
- **More inheritance**: Có thể tạo base classes hoặc interfaces nếu có nhiều common functionality
- **Dependency injection**: Có thể sử dụng DI framework thay vì manual injection
- **Abstract base classes**: Có thể tạo abstract base class cho các entities nếu có nhiều shared behavior

3.7.5.4 Kết luận

Dự án đã áp dụng các nguyên lý OOP một cách hiệu quả, tạo ra một codebase dễ bảo trì, mở rộng và tái sử dụng. Mặc dù có một số cơ hội cải thiện (như sử dụng interfaces cho repositories), codebase hiện tại đã thể hiện tốt các best practices của OOP và tạo nền tảng vững chắc cho việc phát triển và mở rộng hệ thống trong tương lai.

Chương 4 Báo cáo quá trình sử dụng AI

4.1 Khởi tạo và định hướng kiến trúc

4.1.1 Mục tiêu và bối cảnh

Dự án được khởi xướng với mục tiêu xây dựng một ứng dụng Desktop quản lý nhà hàng hiện đại, đóng vai trò như một đồ án học tập mẫu mực. Yêu cầu tiên quyết là sự kết hợp giữa công nghệ mạnh mẽ (Java 17, JavaFX) và một cấu trúc mã nguồn chuẩn nghiệp vụ nhưng không phụ thuộc vào cơ sở dữ liệu bên ngoài để tối ưu hóa tốc độ học tập và triển khai.

4.1.2 Thiết lập nền tảng kỹ thuật cùng AI

Giai đoạn đầu tiên tập trung vào việc định hình "xương sống" cho ứng dụng. Dưới sự tư vấn của AI (đóng vai trò Senior Developer), hệ thống đã thống nhất các tiêu chuẩn:

- **Quản lý dự án:** Sử dụng Maven để quản lý thư viện phụ thuộc và vòng đời sản phẩm.
- **Mô hình kiến trúc:** Áp dụng kiến trúc đa tầng (Layered Architecture). AI đã đề xuất chia tách mã nguồn thành các Package chuyên biệt:
    - model: Chứa các thực thể dữ liệu (Entities) như Employee, Shift, Inventory.
    - repository: Tầng truy cập dữ liệu, sử dụng In-memory Collections thay vì SQL.
    - service: Tầng chứa các quy tắc nghiệp vụ (Business Logic).
    - controller: Điều khiển tương tác giữa giao diện người dùng (UI) và logic hệ thống.
    - app: Chứa điểm khởi chạy ứng dụng (MainApp).

4.2 Triển khai các module nghiệp vụ cốt lõi

4.2.1 Module Quản lý Nhân sự (Human Resources)

AI đã hỗ trợ xây dựng một bảng điều khiển trung tâm cho phép quản lý hồ sơ nhân viên. Điểm nhấn ở đây là việc sử dụng TableView của JavaFX kết hợp với các ObservableList để tạo ra sự phản hồi giao diện tức thì khi có thay đổi dữ liệu (Thêm/Sửa/Xóa).

4.2.2 Module Quản lý Lịch làm việc (Shift Scheduling)

Đây là phần có độ phức tạp cao nhất về mặt logic và giao diện:

- **Giao diện trực quan:** Thay vì các bảng nhập liệu khô khan, AI đã giúp thiết kế một hệ thống lịch làm việc phân chia theo ca (Sáng, Chiều, Tối).
- **Phân loại vai trò:** Hệ thống cho phép gán màu sắc riêng biệt cho từng loại nhân viên (ví dụ: Đầu bếp màu đỏ, Thu ngân màu xanh) ngay trên lịch làm việc, giúp người quản lý có cái nhìn tổng quát về sự phân bổ nhân sự.

4.2.3 Module Quản lý Kho hàng và Đơn hàng (Inventory & Orders)

- **Logic cảnh báo:** AI đã triển khai thuộc tính minimumQuantity (số lượng tối thiểu). Khi nguyên liệu trong kho chạm ngưỡng này, hệ thống sẽ tự động hiển thị cảnh báo, giúp nhà hàng chủ động trong việc nhập hàng.
- **Luồng dữ liệu:** Các giao dịch nhập xuất kho được thiết kế để liên kết chặt chẽ với module Đơn hàng, tạo thành một quy trình khép kín từ lúc nhập nguyên liệu đến khi bán sản phẩm.

4.3 Xử lý sự cố và tối ưu hóa hệ thống (Debugging)

Quá trình làm việc với AI đạt đến đỉnh cao khi giải quyết các vấn đề phát sinh trong thực tế triển khai:

4.3.1 Khắc phục lỗi môi trường thực thi (Runtime Issues)

Khi chạy ứng dụng lần đầu thông qua Maven, dự án gặp lỗi không tìm thấy Module JavaFX. AI đã thực hiện một loạt các bước chẩn đoán:

- Phân tích tệp pom.xml và cập nhật phiên bản javafx-maven-plugin.
- Cấu hình tham số --add-modules để máy ảo Java (JVM) có thể nhận diện chính xác các thư viện đồ họa.
- Xây dựng tệp script run.bat để người dùng có thể khởi động ứng dụng mà không cần gõ các lệnh terminal phức tạp.

4.3.2 Bài toán đồng bộ hóa dữ liệu (The Data Inconsistency Challenge)

Một lỗi logic nghiêm trọng xuất hiện: Khi người dùng thêm nhân viên ở Tab "Employees", dữ liệu này không xuất hiện trong danh sách lựa chọn của Tab "Shifts".

- **Nguyên nhân:** AI chỉ ra rằng mỗi Controller đang khởi tạo một bản sao độc lập của Repository, dẫn đến việc dữ liệu bị chia cắt thành nhiều "ốc đảo" khác nhau.
- **Giải pháp (Design Pattern):** AI đã hướng dẫn áp dụng Singleton Pattern cho tầng Repository. Bằng cách chỉ cho phép tồn tại duy nhất một Instance (thể hiện) của dữ liệu trong suốt vòng đời ứng dụng, mọi thay đổi ở bất kỳ module nào cũng sẽ được cập nhật đồng nhất trên toàn hệ thống.

4.4 Nâng cao trải nghiệm người dùng (UI/UX)

Không chỉ dừng lại ở mã nguồn, AI còn tư vấn cách tinh chỉnh trải nghiệm người dùng:

- **Tối ưu hóa Form nhập liệu:** Thêm các tính năng tự động xóa trắng form (Clear) sau khi lưu thành công và tự động lấy nét (Focus) vào ô nhập liệu đầu tiên.
- **Xử lý lỗi người dùng:** Thêm các hộp thoại thông báo (Alert Dialog) khi người dùng nhập sai định dạng số hoặc để trống các trường bắt buộc, giúp ứng dụng trở nên "mềm mại" và dễ sử dụng hơn.

- **Sơ đồ hóa:** AI hỗ trợ xuất sơ đồ lớp (Class Diagram) thông qua **PlantUML**, giúp người phát triển có cái nhìn tổng thể về mối quan hệ giữa các đối tượng trong hệ thống.

4.5 Tổng kết và bài học kinh nghiệm

4.5.1 Hiệu quả từ việc sử dụng AI

Việc sử dụng AI trong dự án này không chỉ đơn thuần là "sinh mã" (code generation), mà là một quá trình tương tác liên tục:

- **Tư duy hệ thống:** Người dùng học được cách tổ chức code theo chuẩn doanh nghiệp.
- **Tốc độ:** Rút ngắn thời gian từ ý tưởng đến sản phẩm thực tế từ vài tuần xuống còn vài ngày.
- **Kỹ năng xử lý lỗi:** Học được phương pháp chẩn đoán lỗi logic và lỗi môi trường một cách bài bản.

4.5.2 Hướng phát triển tương lai

Dựa trên nền tảng vững chắc đã xây dựng cùng AI, dự án có thể mở rộng theo các hướng:

- Kết nối với cơ sở dữ liệu thực (MySQL/PostgreSQL) thông qua tầng Repository hiện có.
- Tích hợp hệ thống báo cáo doanh thu bằng biểu đồ (Charts).
- Xây dựng tính năng in hóa đơn trực tiếp từ ứng dụng.

Kết luận: Quá trình phát triển dự án này là minh chứng cho sự cộng tác hiệu quả giữa con người và trí tuệ nhân tạo, tạo ra một sản phẩm công nghệ vừa đảm bảo tính học thuật, vừa có tính ứng dụng cao.

Chương 5 Kết luận và hướng phát triển

5.1 Kết luận

Sau quá trình nghiên cứu, thiết kế và phát triển, nhóm đã hoàn thành phiên bản đầu tiên của **Ứng dụng quản lý nhà hàng (Restaurant Management System)** – một hệ thống desktop được xây dựng bằng ngôn ngữ lập trình Java, sử dụng JavaFX cho giao diện người dùng và áp dụng chặt chẽ các nguyên lý lập trình hướng đối tượng (OOP) cùng kiến trúc phân lớp chuyên nghiệp.

Hệ thống đã triển khai thành công bốn module cốt lõi với đầy đủ các chức năng CRUD (Create, Read, Update, Delete), kiểm tra dữ liệu hợp lệ và trải nghiệm người dùng mượt mà:

- **Quản lý nhân viên:** hỗ trợ thêm, sửa, xóa, tìm kiếm và hiển thị danh sách nhân viên một cách khoa học.
- **Quản lý ca làm việc:** lập lịch ca, kiểm tra xung đột lịch trình, xem lịch theo ngày/tuần/nhân viên.
- **Quản lý kho nguyên vật liệu:** theo dõi tồn kho, nhập/xuất kho, ghi nhận lịch sử giao dịch, cảnh báo tự động khi tồn kho thấp.
- **Quản lý đặt bàn:** đặt bàn, hủy đặt bàn, kiểm tra tính khả dụng, xem danh sách đặt bàn theo ngày và trạng thái.

Tất cả các chức năng đều được thiết kế với giao diện trực quan, sử dụng các thành phần JavaFX hiện đại (TableView, DatePicker, ComboBox, ObservableList, TabPane, v.v.), đảm bảo dễ sử dụng ngay cả với những người không rành công nghệ. Dữ liệu được lưu trữ in-memory (danh sách ArrayList/ObservableList) giúp hệ thống hoạt động nhanh, nhẹ và phù hợp cho mục đích học tập, demo.

5.2 Kinh nghiệm rút ra

Thông qua dự án này, nhóm đã tích lũy được những kinh nghiệm quý báu:

- **Kỹ năng thiết kế hệ thống:** Hiểu rõ tầm quan trọng của việc thiết kế sơ đồ lớp (UML) trước khi viết mã. Một cấu trúc dữ liệu tốt giúp việc xử lý các mối

quan hệ phức tạp (như giữa Employee và Shift hay Order và Item) trở nên mạch lạc hơn.

- **Tư duy giải quyết vấn đề:** Học cách xử lý các xung đột dữ liệu, quản lý trạng thái tồn kho thực tế và xử lý logic tính toán tài chính chính xác đến từng đơn vị nhỏ nhất.
- **Kỹ năng làm việc nhóm:** Cải thiện khả năng phối hợp, chia sẻ module và quản lý tiến độ công việc để đảm bảo các phân hệ (Order, Inventory, HR) có thể kết nối với nhau một cách thống nhất.

5.3 Hướng phát triển trong tương lai

Để đưa ứng dụng trở thành một giải pháp quản lý nhà hàng toàn diện, nhóm dự kiến phát triển thêm các phiên bản nâng cao với các hướng sau:

- **Tích hợp cơ sở dữ liệu thực tế**
  - Chuyển từ lưu trữ in-memory sang sử dụng cơ sở dữ liệu quan hệ như MySQL, SQLite hoặc PostgreSQL.
  - Sử dụng JPA/Hibernate hoặc JDBC để kết nối và quản lý dữ liệu bền vững.
- **Mở rộng module kinh doanh**
  - Thêm module Quản lý thực đơn và Đơn hàng: xây dựng menu món ăn, tạo đơn hàng, giỏ hàng, tính tổng tiền.
  - Triển khai Tính tiền và in hóa đơn: hỗ trợ in hóa đơn trực tiếp qua máy in nhiệt, xuất file PDF.
- **Báo cáo và thống kê**
  - Báo cáo doanh thu theo ngày/tuần/tháng.
  - Thống kê tồn kho, nguyên liệu sắp hết, lượng nhập/xuất.
  - Phân tích xu hướng đặt bàn, giờ cao điểm.
- **Hệ thống xác thực và phân quyền**
  - Thêm tính năng đăng nhập/đăng xuất.
  - Phân quyền theo vai trò: chủ nhà hàng (toàn quyền), quản lý (quyền hạn chế), nhân viên (chỉ xem và cập nhật một phần).
- **Tích hợp thiết bị ngoại vi**

- Kết nối với máy in hóa đơn, máy quét mã vạch.
- Hỗ trợ thanh toán trực tuyến (QR code, ví điện tử).

– **Phiên bản đa nền tảng**
  - Phát triển phiên bản web (Spring Boot + React/Angular).
  - Ứng dụng di động (Flutter hoặc Android native) để nhân viên và khách hàng sử dụng.

– **Ứng dụng trí tuệ nhân tạo**
  - Dự đoán nhu cầu nguyên liệu dựa trên lịch sử bán hàng.
  - Gợi ý món ăn phổ biến hoặc tối ưu hóa lịch ca làm việc.

Tài liệu tham khảo

Cursor - The AI Code Editor. (2024). Cursor.com. https://cursor.com/

Google. (2025). *Gemini*. Gemini.google.com; Google. https://gemini.google.com/app

OpenAI. (2025). *ChatGPT*. ChatGPT; OpenAI. https://chatgpt.com/