Project Report on Smart Food Delivery Tracker

MSc in Software Design with Artificial Intelligence
Technological University of the Shannon (TUS)

Author: Bhaumik Shyam Birje (A00335935)

1. Introduction

The Smart Food Delivery Tracker is a Java application designed to manage the complete lifecycle of food delivery orders in a restaurant/delivery service environment. The system facilitates interaction between three main actors: customers who place orders, delivery managers who coordinate operations, and delivery agents who execute deliveries.

The application was developed using Java 24, leveraging both fundamental and advanced object-oriented programming features. The domain was chosen for its practical relevance and complexity, providing realistic scenarios for demonstrating OOP principles including order placement, status tracking, agent assignment, and delivery completion. The system architecture follows a service-oriented approach with clear separation between data models (records), business logic (service layer), and user interactions, ensuring maintainability and extensibility.

2. User Stories Completed

US-001: As a customer, I can place an order by selecting items from a menu and providing my delivery address
US-002: As a customer, I receive immediate confirmation with order details when my order is placed
US-003: As a delivery manager, I can assign available delivery agents to pending orders
US-004: As a customer, I can track my order status (PLACED → PREPARING → DISPATCHED → DELIVERED)
US-005: As a delivery agent, I can accept and deliver assigned orders
US-006: As a delivery manager, I can view lists of active and completed orders
US-007: As a customer, I receive a bill with the total price after successful delivery
US-008: As an administrator, I can manage the menu (add, remove, search items)
US-009: As a delivery agent, my availability status is automatically managed by the system
US-010: As a customer, I receive an estimated delivery time when placing an order

3. Evaluation

3.1 Adherence to Project Brief

Fundamental Features:

All fundamental language features were successfully implemented and demonstrated:

- **Classes**: Multiple classes (Customer, DeliveryAgent, DeliveryManager, Menu) with proper encapsulation
- **this() vs this.**: Constructor chaining in User class using `this()` to call overloaded constructors; `this.` used throughout to distinguish fields from parameters
- **Method overloading**: `updateStatus(Order o)` and `updateStatus(Order o, OrderStatus s)` in DeliveryManager
- **Varargs**: `addItems(Item... items)` in Menu class for flexible item addition
- **LVTI**: `var` keyword used extensively (e.g., `var order = new Order(...)`, `var activeOrders = manager.viewActiveOrders()`)
- **Encapsulation**: All fields private with public getter methods providing controlled access
- **Interfaces**: Service interfaces implemented for manager operations
- **Inheritance**: User sealed class hierarchy with Customer and DeliveryAgent subclasses
- **Overriding and polymorphism**: `getDetails()` method overridden in subclasses, allowing polymorphic behavior
- **super() vs super.**: `super()` calls parent constructor in subclasses; `super.getDetails()` accesses parent methods
- **Exceptions**: IllegalArgumentException for invalid inputs, IllegalStateException for business rule violations, comprehensive null checking
- **Enums**: OrderStatus enum with four states providing type-safe status management
- **Arrays**: Used in varargs implementation and internal collection operations
- **Java Core API**: Extensive use of String/StringBuilder for text processing, List/ArrayList for collections, LocalDateTime/Duration for time handling

Advanced Features:

All advanced features were successfully demonstrated:

- **Call-by-value and defensive copying**: Order record uses `List.copyOf(items)` to prevent external modification; DeliveryManager returns `new ArrayList<>(activeOrders)` to protect internal state
- **private, default, static interface methods**: Service interfaces include default methods for common operations and private helper methods
- **Records**: Order and Item implemented as immutable records with automatic constructor, getters, equals(), hashCode(), and toString()
- **Custom immutable type**: Order record with all final fields and defensive copying ensures complete immutability
- **Lambdas (Predicate)**: Filtering available agents with `agents.stream().filter(agent -> agent.available())`, order filtering, and stream operations throughout

- **'final' and 'effectively final'**: Lambda expressions capture only effectively final variables; documented examples of compilation errors when attempting to modify captured variables
- **Method references**: `Item::price`, `Order::id`, `System.out::println` used for cleaner code
- **Switch expressions**: Modern switch for status messages: `switch(order.status()) { case ORDER_PLACED -> "Received"; ... }`
- **Pattern matching**: Used with sealed classes and record patterns for destructuring
- **Sealed classes**: User sealed class with `permits Customer, DeliveryAgent` clause, both subclasses are final

3.2 Problems Encountered and Solutions

Problem 1: Order Immutability with Status Updates

The Order record is immutable, but order status needs to change throughout its lifecycle. Initially, I considered adding a mutable status field, which would violate immutability principles. **Solution**: Implemented a pattern where status updates return new Order instances with the updated status, preserving immutability. This required refactoring the DeliveryManager to always replace orders in collections rather than modifying them in place. **Learning**: Immutability requires a different mental model but provides thread-safety and predictability benefits.

Problem 2: Sealed Class Design Decisions

Determining which classes should extend User and ensuring the permits clause was correct proved challenging. Initially, I considered making DeliveryAgent non-final to allow specializations (e.g., BicycleAgent, MotorbikeAgent). **Solution**: Kept the hierarchy simple with only Customer and DeliveryAgent as final classes, using composition (vehicleNo field) rather than inheritance for agent variations. **Learning**: Sealed classes work best with small, well-defined hierarchies where all subtypes are known at design time.

Problem 3: Defensive Copying Performance Concerns

Creating defensive copies of item lists on every Order creation and retrieval could impact performance with large orders. **Solution**: Used `List.copyOf()` which creates immutable snapshots efficiently, and documented that the API returns unmodifiable views. Measured negligible performance impact for typical order sizes (< 50 items). **Learning**: Premature optimization is the root of all evil; immutability benefits outweigh minor performance costs in most business applications.

Problem 4: Java 24 Preview Feature Configuration

IntelliJ IDEA initially showed warnings for Java 24 preview features, and Maven builds failed. **Solution**: Updated IDE to latest version (2024.3), explicitly enabled preview features in Maven compiler plugin configuration, and added `--enable-preview` flag to both compiler and runtime arguments. Had to ensure the JDK 24 installation was properly configured in IDE project structure. **Learning**: Preview features require explicit opt-in at multiple levels (compiler, runtime, IDE) to ensure consistency.

3.3 Summary

The project successfully demonstrates comprehensive mastery of Java OOP principles, from fundamental concepts to cutting-edge Java 24 features. All required language features were implemented with practical, working examples. The food delivery domain provided realistic complexity for showcasing encapsulation, inheritance, polymorphism, and modern Java idioms including records, sealed classes, and pattern matching.

The final application is functional, maintainable, and well-documented, with clean separation of concerns and adherence to SOLID principles. The use of Java 24 allowed exploration of the latest language enhancements while maintaining backward compatibility with required Java 21 features. The consistent Git repository history demonstrates professional development practices with incremental, meaningful commits.

Code quality is high with proper encapsulation, comprehensive error handling, and extensive use of immutability for thread-safety. The combination of defensive copying, sealed classes, and records creates a robust, type-safe application that minimizes potential runtime errors through compile-time guarantees.