SRS-DLD

S/W Detailed Level Design

Project Name	Photo Studio Management System		
Block Name	Release 3		
Author	Team	**Approver**	Anna Kyselova
Team	Cebanu Dan (@Nik0gda) Nikita Springis (@nikitaspringis21) Manana Bebia (@Mabebi29) Anna Sukhanishvili (@annasulkh)		

Contents

■ Revision History

Version	Date	Revised contents	Author	Approver
1.0	23.09.2025	Release 1	Team	Anna Kyselova
1.1	26.10.2025	Release 2	Team	Anna Kyselova
1.2	24.11.2025	Release 3	Team	Anna Kyselova

■ Terms and Abbreviations

Term	Description
DIP	Dependency Inversion Principle
RAII	Resource Acquisition Is Initialization
UML	Unified Modeling Language
SRP	Single Responsibility Principle
OCP	Open/Closed Principle

■ References

1. SW Requirements Specification

1. Overview

This document provides the detailed technical design for the Photo Studio Management System, a C++ application that automates the workflow of a photo studio. The system manages client orders for photo printing and film developing services, tracks employee responsibilities across three roles (Receptionist, Photographer, Administrator), manages consumable inventory, and generates comprehensive reports.

The system supports both regular and express orders (with 25% surcharge), tracks consumable usage during photo processing, and provides end-of-day reporting capabilities. The design follows SOLID principles with clear separation of concerns, dependency injection, and polymorphic behavior to ensure maintainability and extensibility.

Key stakeholders include studio receptionists who create orders, photographers who process orders and track consumables, administrators who manage inventory, and studio management who review reports. The system replaces manual paper-based tracking with an automated console-based application.

2. System Overview / Architectural Context

The system follows a layered architecture with clear separation between presentation, business logic, and data management:

Presentation Layer:

- ConsoleDisplay: Handles all user interface output
- IDisplay interface: Abstracts display functionality for testing and future UI implementations

Business logic layer:

- Manager Classes: OrderManager, ConsumableManager, ReportManager
- Employee Classes: Receptionist, Photographer, Administrator (polymorphic hierarchy)
- Domain Services: Order processing, inventory management, report generation

Domain layer:

- Core Entities: Order/ExpressOrder, Client, Service, Consumable
- Value Objects: OrderItem, ConsumableUsage, Report
- Enumerations: OrderStatus, ServiceType, ReportType

Infrastructure layer:

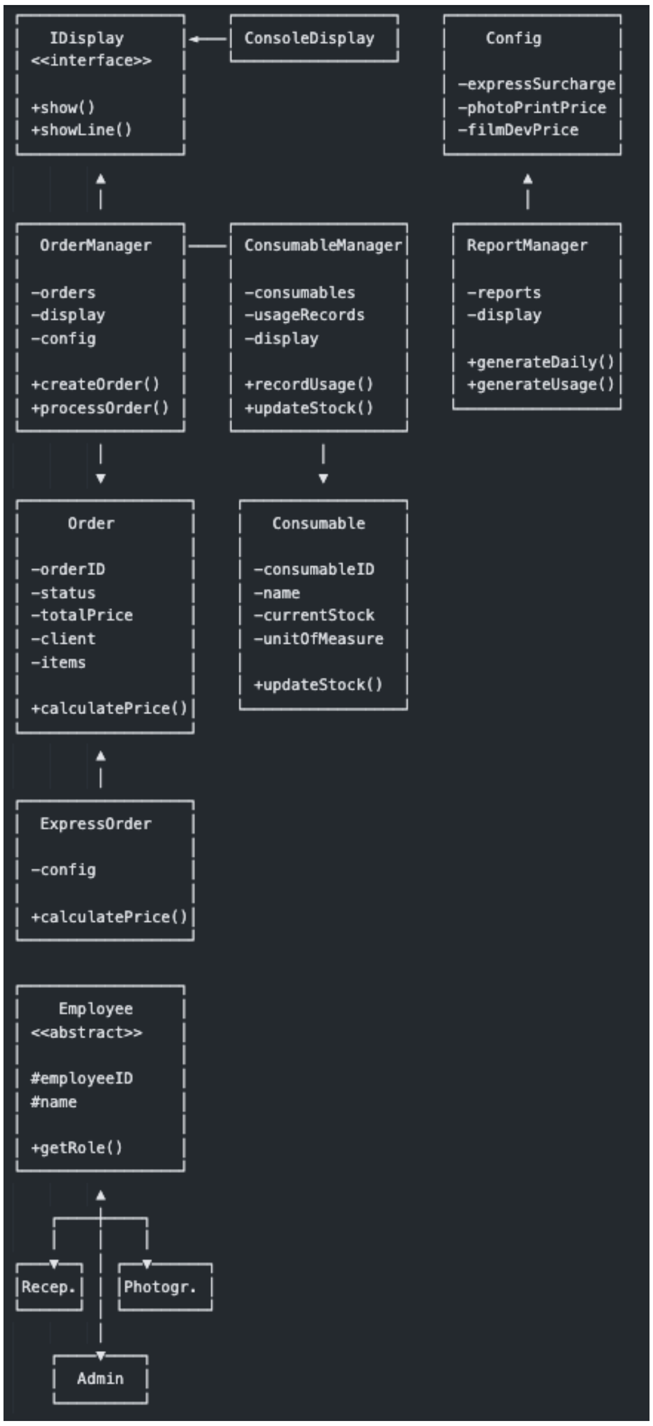
- Configuration: Config class for business rules and pricing
- Types: Centralized type definitions and enumerations

Dependency flow:

Presentation -> Business Logic -> Domain -> Infrastructure

Each layer interacts only with the layer directly below it, ensuring loose coupling and high cohesion. The system uses dependency injection throughout, with the IDisplay interface enabling easy testing and future UI changes.

3. UML Class Diagram (Technical Design)



4. Class Specifications

Class	Type	Description	Attributes	Methods
Order	Entity	Represents a client order for photo services	orderID (string), completionTime (string), status(OrderStatus), totalPrice(double), isPaid(bool), client(Client*), items(vector<OrderItem*>)	calculatePrice(), updateStatus(), recordPayment(), addItem(), getters
ExpressOrder	Entity	Express order with surcharge	config(Config*)	calculatePrice() override
Client	Entity	Represents a studio client	clientID(string), surname(string)	getSurname(), getID()
Employee	Abstract	Base class for all employees	employeeID(string), name(string)	getName(), getID(), getRole()
Receptionist	Employee	Handles order creation and reports	Inherited	createOrder(), generateDailyRevenueReport() getRole()
Photographer	Employee	Processes orders and tracks consumables	Inherited	viewAssignedOrders(), submitConsumablesReport(), getRole()
Administrator	Employee	Manages inventory and reviews reports	Inherited	manageConsumablesStock(), reviewConsumablesReports(), getRole()
OrderManager	Manager	Manages order lifecycle	orders(vector<Order*>), display(IDisplay*), config(Config*)	createOrder(), addItemToOrder(), processOrder(), completeOrder(), calculateTotalRevenue()
Consumable Manager	Manager	Tracks inventory and usage	consumables(vector<Consumable*>),	addConsumable(), recordUsage(),

			usageRecords(vector<ConsumableUsage>), display(IDisplay*)	updateStock(), findConsumableByName()
ReportManager	Manager	Generates and stores reports	reports(vector<Report*>), display(IDisplay*)	generateDailyRevenueReport(), generateConsumablesUsageRepo getAllReports()
Consumable	Entity	Represents studio consumables	consumableID(string), name(string), currentStock(int), unitOfMeasure(string)	updateStock(), getCurrentStock(), getName(), getConsumableID()
Service	Entity	Represents available services	serviceID(string), name(string), basePrice(double), type(ServiceType)	getBasePrice(), getName(), getServiceID(), getType()
OrderItem	Value Obj	Individual items in an order	itemID(string), quantity(int), unitPrice(double)	getSubtotal(), getItemID(), getQuantity(), getUnitPrice()
Config	Config	Business rules and pricing	expressSurchargeRate(double), photoPrintingBasePrice(double), filmDevelopingBasePrice(double)	Getters and setters for all rates ar prices

5. Interfaces and Abstractions

Interface	Purpose	Key Methods	Planned For (Release)
IDisplay	Abstract output interface for UI flexibility	show(message), showLine(message)	Release 2
Employee	Abstract base for polymorphic employee behaviour	getRole()	Release 2
Order	Virtual base for polymorphic pricing	calculatePrice()	Release 1

6. Function Responsibilities

Class	Method	Purpose	Input	Output	Notes
OrderManager	createOrder()	Factory method for order creation	orderID, client, completionTime, isExpress	Order*	Creates regular or express order based on flag
OrderManager	calculateTotalRevenue()	Sums revenue from all paid orders	None	double	Iterates through orders, sums paid totals
ExpressOrder	calculatePrice()	Applies 25% surcharge to base price	None	double	Overrides base implementation
ConsumableManager	recordUsage()	Updates inventory when consumables used	ConsumableUsage	void	Automatically reduces stock
ReportManager	generateDailyRevenueReport()	Creates revenue summary	OrderManager*	void	Aggregates order data
Employee	getRole()	Returns employee type	None	string	Pure virtual, implemented by subclasses

7. Operation Flow

Primary order processing flow:

1. Client Interaction: Client approaches receptionist with service request
2. Order Creation: Receptionist -> OrderManager.createOrder() -> Order/ExpressOrder created
3. Item Addition: OrderManager.addItemToOrder() -> OrderItem created and added
4. Price Calculation: Order.calculatePrice() -> Base price or surcharge applied
5. Order Processing: OrderManager.processOrder() -> Status updated to IN_PROGRESS
6. Photographer Assignment: Photographer.viewAssignedOrders() -> Order retrieved
7. Consumable Usage: Photographer.submitConsumablesReport() -> ConsumableManager.recordUsage()
8. Inventory Update: ConsumableManager updates stock automatically
9. Order Completion: OrderManager.completeOrder() -> Status updated to COMPLETED
10. Payment Processing: OrderManager.recordPayment() -> Payment recorded

End-of-day reporting flow:

1. Revenue Report: Receptionist -> ReportManager.generateDailyRevenueReport()
2. Usage Report: Administrator -> ReportManager.generateConsumablesUsageReport()
3. Report Storage: Reports stored in ReportManager for history

Data flow:

ConsoleDisplay <-> Manager Classes <-> Domain Entities <-> Configuration

8. Enumerations & Constants

Name	Value / Type	Description
OrderStatus	Enum class	PENDING, IN_PROGRESS, COMPLETED, CANCELLED
ServiceType	Enum class	PHOTO_PRINTING, FILM_DEVELOPING
ReportType	Enum class	DAILY_REVENUE, CONSUMABLES_USAGE
EXPRESS_SURCHARGE_RATE	0.25 (double)	25% surcharge for express orders
PHOTO_PRINTING_BASE_PRICE	15.99 (double)	Base price for photo printing service
FILM_DEVELOPING_BASE_PRICE	25.50 (double)	Base price for film developing service

9. Validation Rules & Future Work

Rule / Planned Feature	Description	Target Release
Input Validation	Validate UI inputs (non-empty strings, numeric ranges, supported formats)	Release 3
Entity invariants	Validate domain constructor invariants (Order id, client pointer, non-empty time, positive quantities)	Release 3

Exception handling policy	Centralized exception types	Release 3
Repository ownership and memory model	Repository allocates and deallocates all objects	Release 4
Dynamic array growth	Manual dynamic array with amortized resizing, correct reallocation and pointer update semantics.	Release 4

10. Traceability Matrix

Requirement (SRS)	**Class / Method (DLD)**
“Client places order with receptionist”	Receptionist::createOrder(), OrderManager::createOrder()
“Record client surname and completion time”	Client class, Order constructor
“Express orders have 25% surcharge”	ExpressOrder::calculatePrice(), Config::getExpressSurchargeRate()
“Payment made upon completion”	Order::recordPayment(), OrderManager::recordPayment()
“Photographer uses consumables”	Photographer::submitConsumablesReport() ConsumableManager::recordUsage()
“Administrator accounts for consumables”	Administrator::manageConsumablesStock(), ConsumableManager
“Receptionist reports on revenue”	Receptionist::generateDailyRevenueReport(), ReportManager::generateDailyRevenueReport()
“Photographer reports consumed materials”	Photographer::submitConsumablesReport(), ReportManager::generateConsumablesUsageReport()

Requirement (SRS)	Class / Method (DLD)
“Two forms: client and photographer”	Order creation workflow, Order status tracking

11. Code Structure and File Mapping

Class	File
Order	src/orders/Order.cpp / src/orders/Order.h
ExpressOrder	src/orders/ExpressOrder.cpp / src/orders/ExpressOrder.h
Client	src/entities/Client.cpp / src/entities/Client.h
Employee	src/employees/Employee.cpp / src/employees/Employee.h
Receptionist	src/employees/Receptionist.cpp / src/employees/Receptionist.h
Photographer	src/employees/Photographer.cpp / src/employees/Photographer.h
Administrator	src/employees/Administrator.cpp / src/employees/Administrator.h
OrderManager	src/managers/OrderManager.cpp / src/managers/OrderManager.h
ConsumableManager	src/managers/ConsumableManager.cpp / src/managers/ConsumableManager.h
ReportManager	src/managers/ReportManager.cpp / src/managers/ReportManager.h
Consumable	src/entities/Consumable.cpp / src/entities/Consumable.h
Service	src/entities/Service.cpp / src/entities/Service.h
OrderItem	src/entities/OrderItem.cpp / src/entities/OrderItem.h
ConsumableUsage	src/entities/ConsumableUsage.cpp / src/entities/ConsumableUsage.h
Report	src/entities/Report.cpp / src/entities/Report.h
Config	src/config/Config.cpp / src/config/Config.h
IDisplay	src/interfaces/IDisplay.h

Class	File
ConsoleDisplay	src/implementations/ConsoleDisplay.cpp / src/implementations/ConsoleDisplay.h
Types	src/types/Types.h
Main Application	src/main.cpp

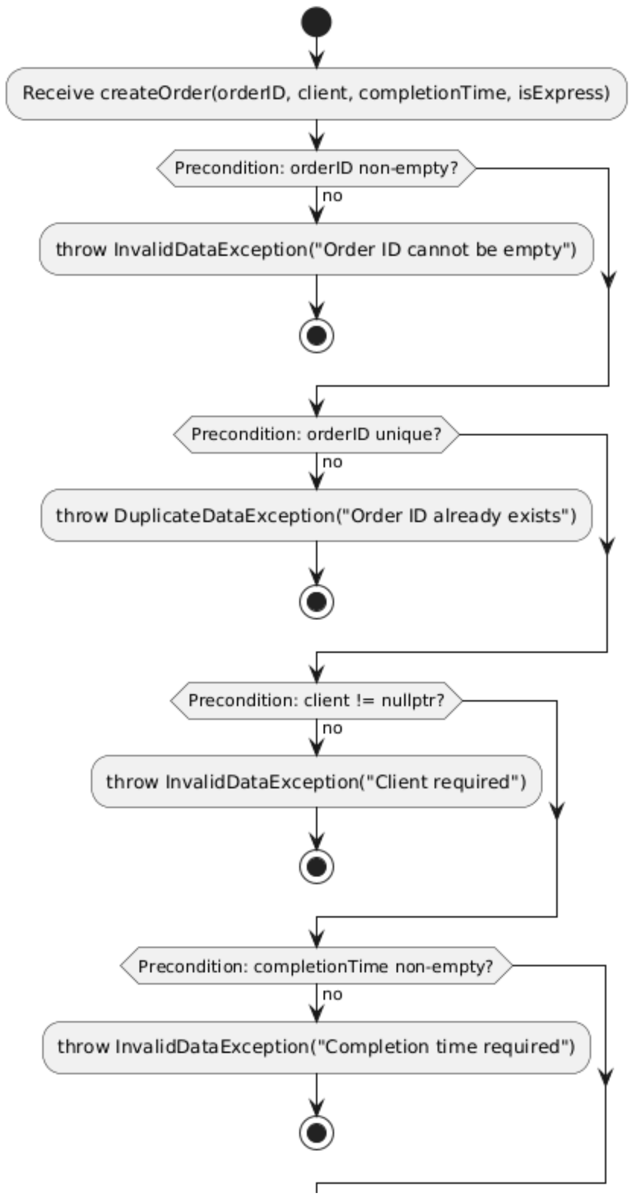
12. Validation Rules & Preconditions/Postconditions

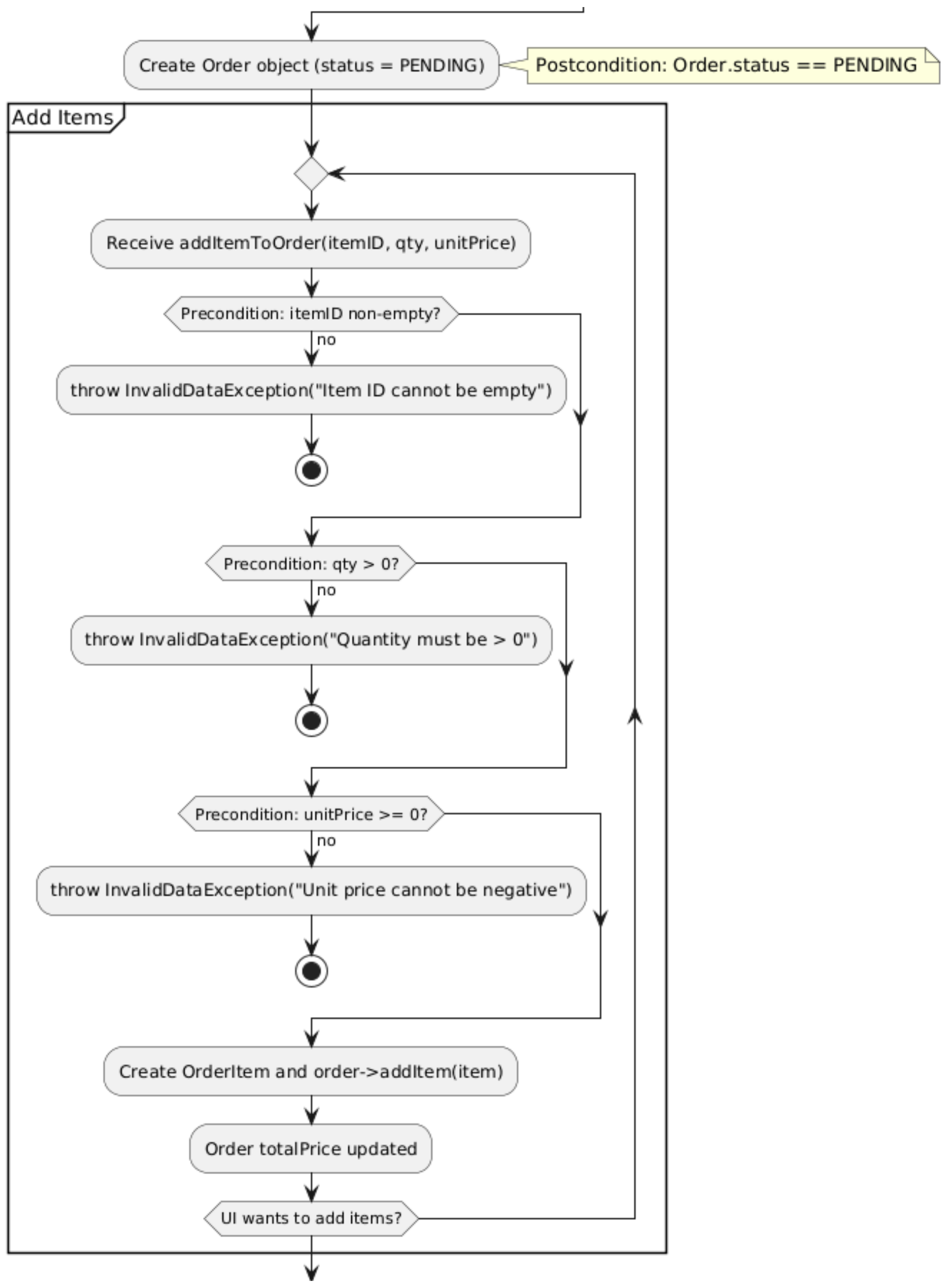
Class::method	Preconditions	Postconditions	Validation Level	Explanation
OrderManager::createOrder (header)	orderID non-empty; orderID not duplicate; client != nullptr; completionTime non-empty	returned Order exists, status == PENDING, order in manager list	Logic	Validation implemented in OrderManager::validateOrderCreation. UI (e.g. Receptionist::createOrder) may show messages but logic enforces invariants.
OrderManager::addItemToOrder	order != nullptr; quantity > 0; unitPrice >= 0; itemID non-empty	item appended to order; order totalPrice updated (via Order::addItem)	Logic / Repository	Uses validateOrderExists and validateOrderItem. Order::addItem throws on null item (logic).
OrderManager::processOrder	order != nullptr; order.status == PENDING	order.status == IN_PROGRESS	Logic	validateOrderExists + validateOrderStatus enforce preconditions; postcondition asserted after order->updateStatus.
OrderManager::completeOrder	order != nullptr; order.status == IN_PROGRESS	order.status == COMPLETED; price >= 0	Logic	Calls order->updateStatus and order->calculatePrice(). Postconditions checked and ValidationException thrown on violation.
OrderManager::recordPayment	order exists; order.status == COMPLETED; order has items; totalPrice > 0	order.isPaid == true	Logic	Precondition checks + call to order->recordPayment. Postcondition validated.
ConsumableManager::addConsumable (header)	consumable != nullptr; consumable fields valid; consumableID unique	consumable added to manager list	Logic / Repository	Uses validateConsumable which checks ID/name/duplicate (logic) before appending to repository vector.

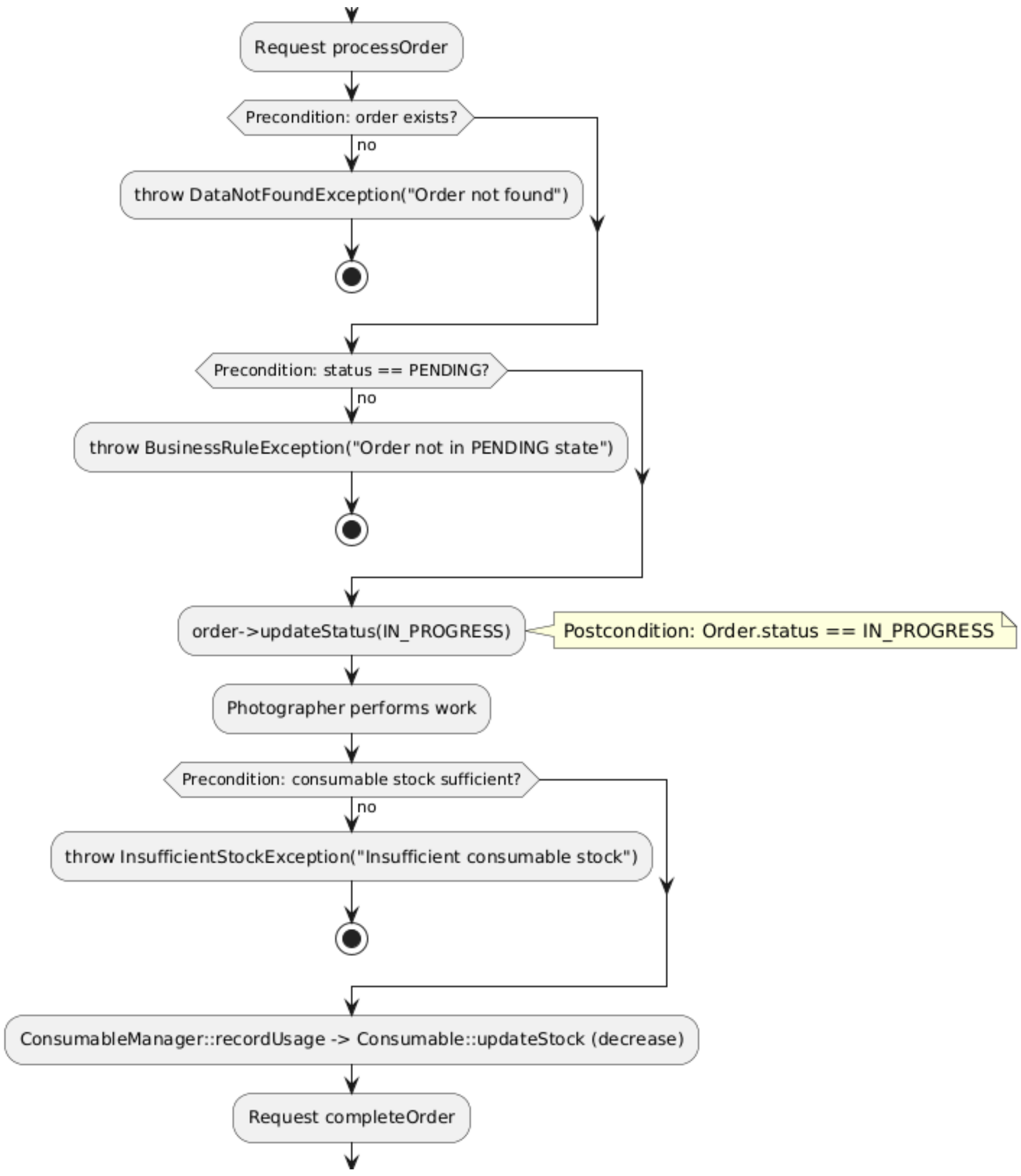
ConsumableManager::recordUsage	usage.consumableName non-empty; usage.quantity > 0; consumable exists	usage appended to usageRecords; consumable stock decreased	Logic / Repository	validateConsumableUsage ensures preconditions; then updates repository (Consumable::updateStock).
ConsumableManager::updateStock	consumableName non-empty; consumable exists; newStock >= 0	consumable currentStock updated	Logic / Repository	validateStockUpdate checks existence and non-negative result; actual mutation via Consumable::updateStock.
Consumable::Consumable constructor	id non-empty; name non-empty; stock >= 0; unit non-empty	constructed object with valid fields	Repository (entity-level validation)	Constructor throws InvalidDataException on invalid inputs.
ConsumableUsage::ConsumableUsage constructor	id non-empty; name non-empty; qty > 0	constructed usage record	Repository / Logic	Constructor-level validation throws InvalidDataException for bad inputs.
Order::Order constructor	id non-empty; cTime non-empty; client != nullptr	Order created with status PENDING, totalPrice == 0	Repository / Logic	Constructor validates inputs and throws InvalidDataException.
Order::addItem	item != nullptr	item added; totalPrice increased by item ->getSubtotal()	Logic / Repository	Throws InvalidDataException for null item; maintains postcondition via subtotal update.
ReportManager:: generateDailyRevenueReport	orderManager != nullptr	a Report created and stored in reports list; content consistent with OrderManager state	Logic	Uses OrderManager API (OrderManager::getAllOrders) to compute revenue; input validation at logic layer.
ReportManager:: generateConsumablesUsageReport	consumableManager != nullptr	Report created summarizing usageRecords	Logic	Uses created summarizing usageRecords | Logic | Uses [ConsumableManager::getUsageRecords](src/managers/ConsumableManager.cpp).

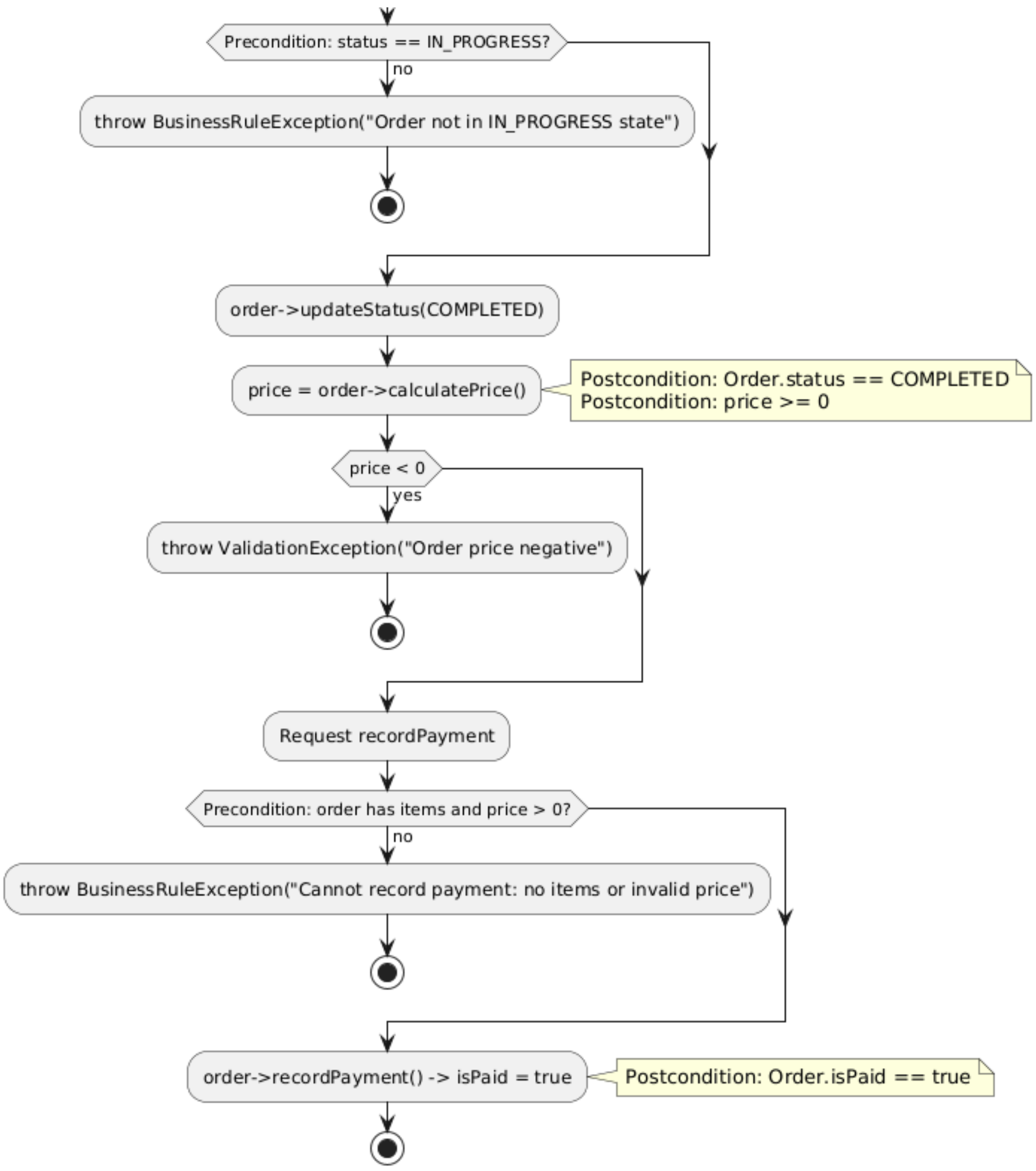
13. Behavioral Models

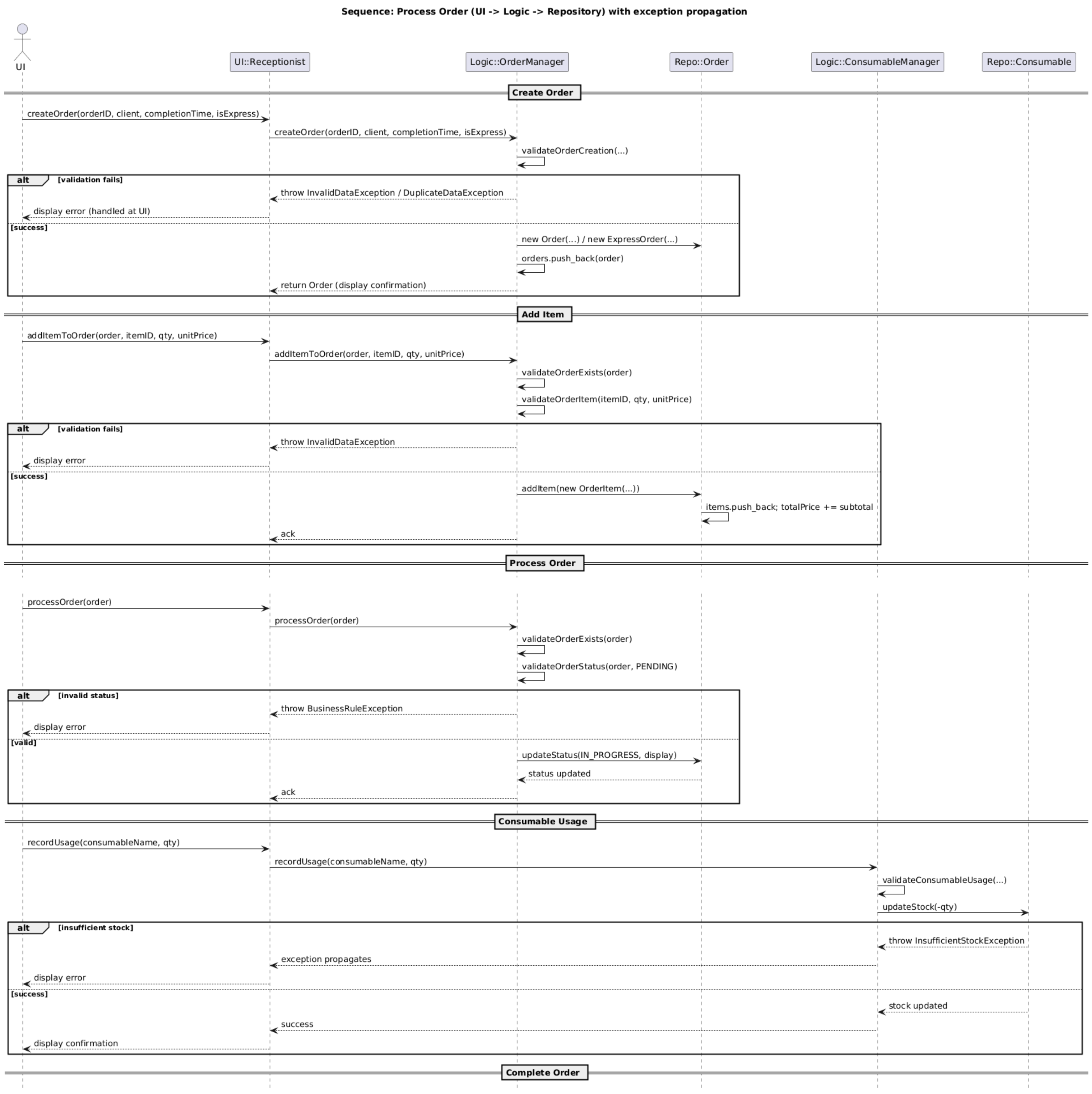
Activity: Process Order — internal logic (preconditions as decisions, postconditions as finals)

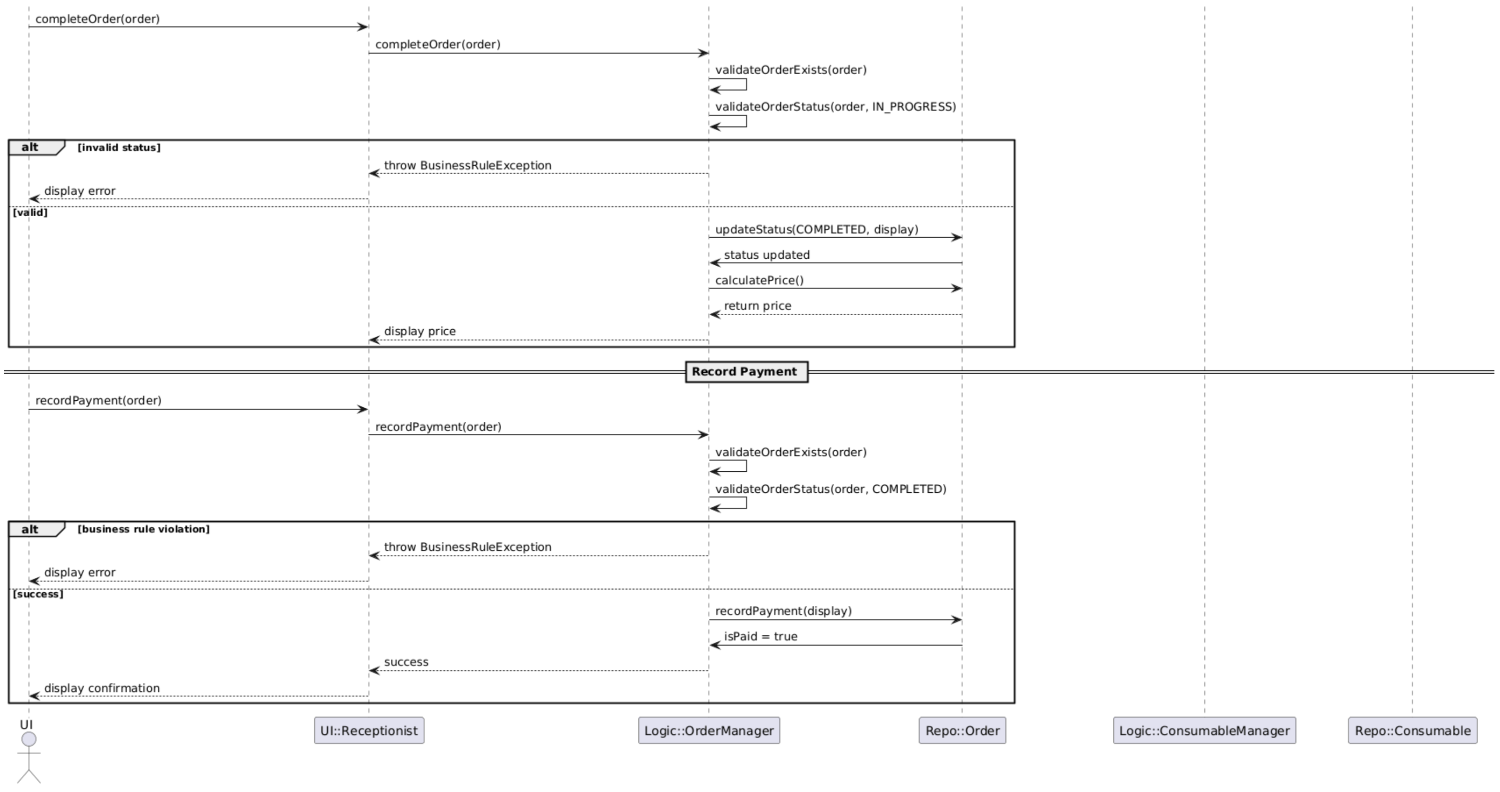

Receive createOrder(orderID, client, completionTime, isExpress)
Precondition: orderID non-empty?
no
throw InvalidDataException("Order ID cannot be empty")
Precondition: orderID unique?
no
throw DuplicateDataException("Order ID already exists")
Precondition: client != nullptr?
no
throw InvalidDataException("Client required")
Precondition: completionTime non-empty?
no
throw InvalidDataException("Completion time required")


Create Order object (status = PENDING)
Postcondition: Order.status == PENDING
Add Items
Receive addItemToOrder(itemID, qty, unitPrice)
Precondition: itemID non-empty?
no
throw InvalidDataException("Item ID cannot be empty")
Precondition: qty > 0?
no
throw InvalidDataException("Quantity must be > 0")
Precondition: unitPrice >= 0?
no
throw InvalidDataException("Unit price cannot be negative")
Create OrderItem and order->addItem(item)
Order totalPrice updated
UI wants to add items?


Request processOrder
Precondition: order exists?
no
throw DataNotFoundException("Order not found")
Precondition: status == PENDING?
no
throw BusinessRuleException("Order not in PENDING state")
order->updateStatus(IN_PROGRESS)
Postcondition: Order.status == IN_PROGRESS
Photographer performs work
Precondition: consumable stock sufficient?
no
throw InsufficientStockException("Insufficient consumable stock")
ConsumableManager::recordUsage -> Consumable::updateStock (decrease)
Request completeOrder


Precondition: status == IN_PROGRESS?
no
throw BusinessRuleException("Order not in IN_PROGRESS state")
order->updateStatus(COMPLETED)
price = order->calculatePrice()
Postcondition: Order.status == COMPLETED
Postcondition: price >= 0
price < 0
yes
throw ValidationException("Order price negative")
Request recordPayment
Precondition: order has items and price > 0?
no
throw BusinessRuleException("Cannot record payment: no items or invalid price")
order->recordPayment() -> isPaid = true
Postcondition: Order.isPaid == true


Sequence: Process Order (UI -> Logic -> Repository) with exception propagation
UI
UI::Receptionist
Logic::OrderManager
Repo::Order
Logic::ConsumableManager
Repo::Consumable
Create Order
createOrder(orderID, client, completionTime, isExpress)
createOrder(orderID, client, completionTime, isExpress)
validateOrderCreation(...)
alt [validation fails]
throw InvalidDataException / DuplicateDataException
display error (handled at UI)
[success]
new Order(...) / new ExpressOrder(...)
orders.push_back(order)
return Order (display confirmation)
Add Item
addItemToOrder(order, itemID, qty, unitPrice)
addItemToOrder(order, itemID, qty, unitPrice)
validateOrderExists(order)
validateOrderItem(itemID, qty, unitPrice)
alt [validation fails]
throw InvalidDataException
display error
[success]
addItem(new OrderItem(...))
items.push_back; totalPrice += subtotal
ack
Process Order
processOrder(order)
processOrder(order)
validateOrderExists(order)
validateOrderStatus(order, PENDING)
alt [invalid status]
throw BusinessRuleException
display error
[valid]
updateStatus(IN_PROGRESS, display)
status updated
ack
Consumable Usage
recordUsage(consumableName, qty)
recordUsage(consumableName, qty)
validateConsumableUsage(...)
updateStock(-qty)
alt [insufficient stock]
throw InsufficientStockException
exception propagates
display error
[success]
stock updated
success
display confirmation
Complete Order



14. Error & Exception Handling Policy

Exception Type	**Thrown By (Layer / Class)**	**Caught At (Layer)**	**Message / what()**	**Default Action (message / stop / retry)**
PhotoStudioException (base)	Base for all custom exceptions (src/exceptions/PhotoStudioExceptions.h)	N/A (base)	custom message via what()	Propagates unless specific handler exists
InvalidInputException	UI layer helpers / input parsing (constructed in UI)	UI (src/main.cpp)	"Invalid input provided by user"	UI shows error and exits with code 1 (see main catch)

InvalidDataException	Logic / entity constructors (e.g., Order::Order, Consumable::Consumable, ConsumableUsage::ConsumableUsage)	Logic or UI (src/main.cpp)	user-friendly message set at throw site	Display message and exit; no automatic retry
ValidationException	Logic postcondition checks (e.g., OrderManager::completeOrder)	UI (src/main.cpp)	message describing pre/postcondition failure	Show message and stop operation; return non-zero
BusinessRuleException	Logic (e.g., invalid status transitions, business rules) (src/managers/OrderManager.cpp, ConsumableManager::validateStockUpdate)	UI (src/main.cpp)	e.g., "Cannot reduce stock below zero"	Show message and stop operation
DataNotFoundException	Repository/Manager lookups (e.g., findConsumableByName in ConsumableManager)	UI (src/main.cpp)	e.g., "Consumable not found: NAME"	Show message and exit / prompt user
InsufficientStockException	Repository/Consumable update ([src/managers/ConsumableManager.cpp], Consumable::updateStock)	UI (src/main.cpp)	e.g., "Insufficient stock for X"	Show message and stop; administrator must restock (no automatic retry)
DuplicateDataException	Logic when duplicate ID detected (e.g., ConsumableManager::validateConsumable, OrderManager::validateOrderCreation)	UI (src/main.cpp)	e.g., "Consumable ID already exists: ID"	Show message; reject creation
std::exception / other std errors	any layer (unexpected)	UI general catch (src/main.cpp)	what() from std::exception	Show generic unexpected-error message and stop

15. Revision History

Date	Version	Change Summary	Author
26.10.2025	1.0	Initial DLD creation	Team
24.11.2025	1.2	Validation rules + error handling for release 3	Team