Convenience Store Management System (CSMS)

Github url :

git@github.com:ziiiimu/CSMS.git

https://github.com/ziiiimu/CSMS#

gourp

Lan Yao (Net ID: ly2238)

Florence Zhao (Net ID: tz2556)

TABLE OF WORK	STUDENT-1	STUDENT-2	STUDENT-3
Project Description	Florence Zhao		
Use Cases Diagram(s) and description	Florence Zhao		
Sequence Diagrams	Florence Zhao		
Class Diagram(s)	Florence Zhao		
Implementation	Florence Zhao		
Conclusion	Florence Zhao		

0. Project Structure and Organization

File Organization:

```
CSMS_Project/
├── src/
│   ├── Product.h                  # Abstract product hierarchy
│   ├── Product.cpp                # Product implementations
│   ├── Customer.h                 # Customer management
```

```
|   ├── Customer.cpp           # Customer implementations
|   ├── Transaction.h          # Transaction processing
|   ├── Transaction.cpp        # Transaction implementations
|   ├── InventoryManager.h     # Inventory management
|   ├── InventoryManager.cpp   # Inventory implementations
|   └── Main.cpp               # Application entry point
|   └── Makefile               # Build configuration
├── uml                        # uml source code
├── out                        # uml png
```

1. Project Summary

The **Convenience Store Management System (CSMS)** is a comprehensive, object-oriented software solution designed to streamline the operations of small to medium-sized retail stores. This enterprise-grade application demonstrates advanced software engineering principles, implementing a sophisticated multi-layered architecture that handles inventory management, customer relationship management, transaction processing, and business analytics.

2. System Overview

Core Modules:

1. ** Inventory Management**
    - Multi-type product hierarchy (Regular, Perishable, Bulk)
    - Automated low-stock alerts and restock recommendations
    - Category-based organization and supplier management
    - Real-time inventory valuation and profitability analysis
2. ** Customer Relationship Management**
    - Four-tier customer classification system
    - Dynamic loyalty points calculation with tier-based multipliers
    - Automatic membership upgrades based on spending thresholds
    - Comprehensive customer analytics and behavior tracking
3. ** Transaction Processing**
    - Multi-item transactions with complex pricing calculations
    - Six payment method support (Cash, Credit/Debit Cards, Mobile, Points, Gift Cards)

    - Advanced discount application (customer-based, promotional, near-expiration)
    - Complete refund processing with inventory restoration

4. ** Business Intelligence**
    - Real-time sales reporting and financial analysis
    - Inventory valuation and profit margin calculations
    - Customer behavior analytics and top customer identification
    - Automated alert systems for critical business metrics

3. System Architecture and Design

Architecture Layers:

```
┌─────────────────────────────────────────┐
│              Presentation Layer         │
│           (ConvenienceStoreApp)        │
├─────────────────────────────────────────┤
│               Business Layer            │
│   (InventoryManager, CustomerDatabase,  │
│    Transaction Processing Logic)        │
├─────────────────────────────────────────┤
│                Domain Layer             │
│     (Product Hierarchy, Customer,      │
│      Transaction, TransactionItem)      │
├─────────────────────────────────────────┤
│               Data Layer                │
│    (In-memory Collections, Future:      │
│     Database Integration Layer)         │
└─────────────────────────────────────────┘
```

```
ConvenienceStoreApp
├── InventoryManager ──→ Product (Abstract)
│   │                    ├── RegularProduct
│   │                    ├── PerishableProduct
│   │                    └── BulkProduct
│   └── ProductCategory (Enum)
├── CustomerDatabase ──→ Customer ──→ CustomerType (Enum)
└── Transaction ──→ TransactionItem ──→ Product
    ├── PaymentMethod (Enum)
    └── TransactionStatus (Enum)
```

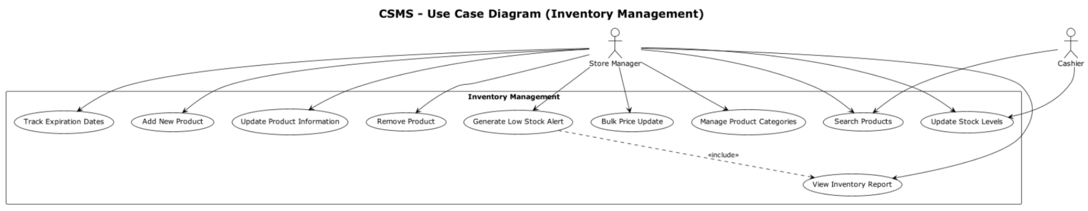
4. UML Diagrams

4.1 Use Case Diagram

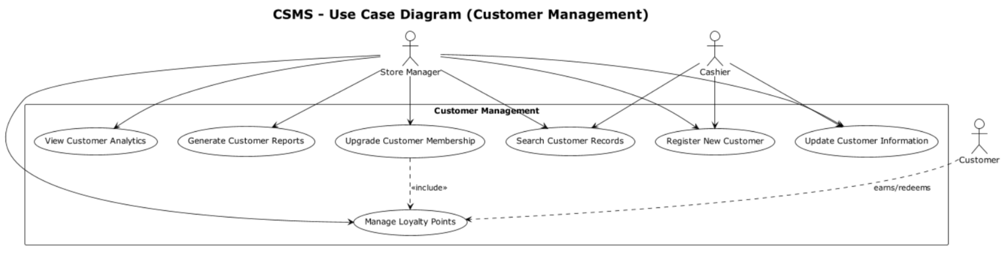
The use case diagram illustrates the comprehensive functionality of the CSMS through **35 distinct use cases** organized into five functional packages:

Use Case Package Organization:

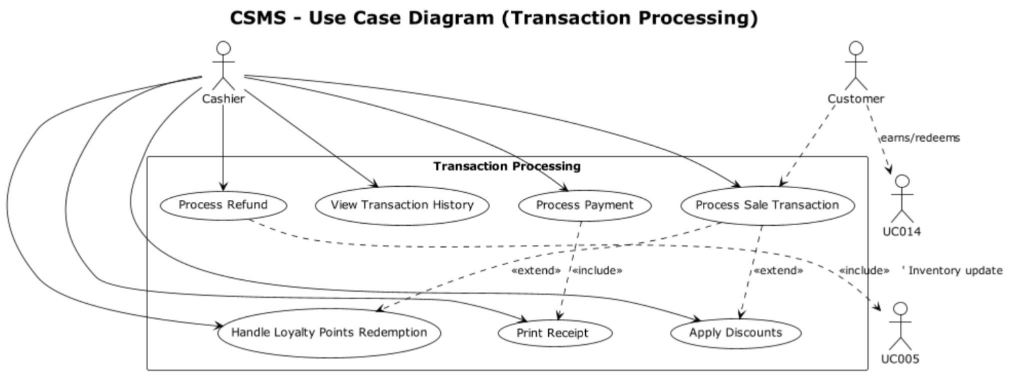
1. **Inventory Management (10 use cases)**



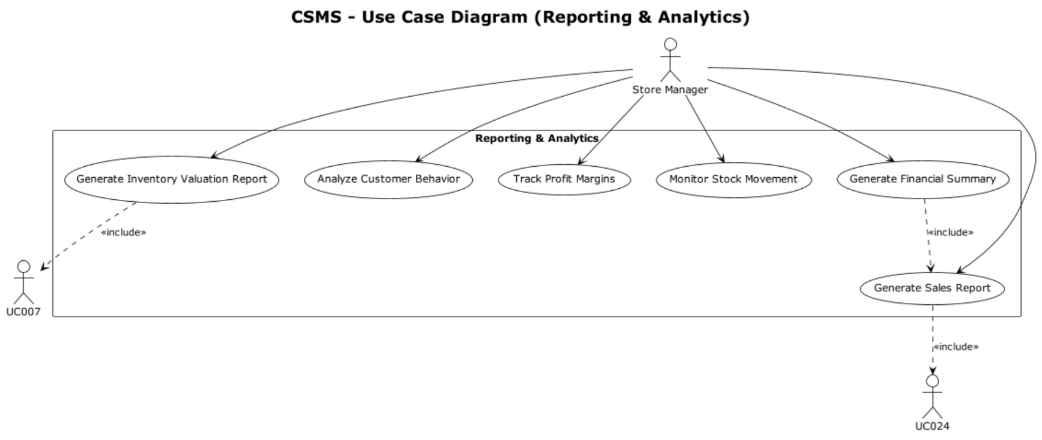
2. **Customer Management (7 use cases)**



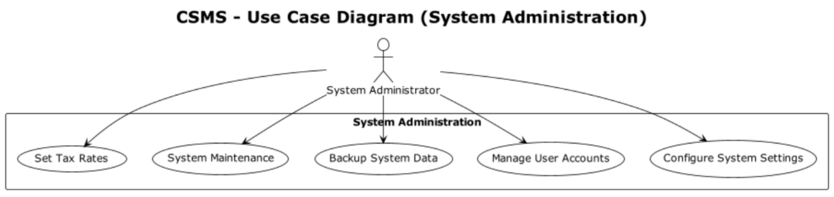
3. **Transaction Processing (6 use cases)**



4. **Reporting & Analytics (7 use cases)**



5. **System Administration (5 use cases)**



Actor Relationships:

- **Cashier:** 13 use cases (operational tasks)
- **Store Manager:** 20 use cases (management and oversight)
- **System Administrator:** 5 use cases (system maintenance)
- **Customer:** 2 use cases (indirect participation)

Use Case Relationships:

- **Include Relationships:** 5 mandatory dependencies
- **Extend Relationships:** 3 optional extensions
- **Generalization:** 2 specialized use cases

Use Case Detail:

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC001	Add New Product
Overview	Store manager adds new products to the inventory system with complete product information
Related use cases	UC002 (Update Product Information), UC009 (Manage Product Categories)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC002	Update Product Information
Overview	Modify existing product details including prices, descriptions, and stock levels
Related use cases	UC001 (Add New Product), UC005 (Update Stock Levels)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC003	Remove Product
Overview	Deactivate or permanently remove products from the system inventory
Related use cases	UC007 (View Inventory Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC004	Search Products

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
Overview	Find products using various criteria such as name, ID, category, or tags
Related use cases	UC018 (Process Sale Transaction)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC005	Update Stock Levels
Overview	Add or reduce inventory quantities for existing products
Related use cases	UC006 (Generate Low Stock Alert), UC022 (Process Refund)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC006	Generate Low Stock Alert
Overview	Automatically detect products with stock levels below minimum thresholds and generate alerts
Related use cases	UC005 (Update Stock Levels), UC007 (View Inventory Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC007	View Inventory Report
Overview	Generate comprehensive inventory reports including stock levels, values, and analytics
Related use cases	UC006 (Generate Low Stock Alert), UC027 (Generate Inventory Valuation Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC008	Bulk Price Update

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
Overview	Update prices for multiple products simultaneously based on categories or criteria
Related use cases	UC002 (Update Product Information), UC009 (Manage Product Categories)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC009	Manage Product Categories
Overview	Organize products into categories and manage category-specific settings and rules
Related use cases	UC001 (Add New Product), UC008 (Bulk Price Update)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC010	Track Expiration Dates
Overview	Monitor expiration dates for perishable products and apply automatic discounts when near expiration
Related use cases	UC006 (Generate Low Stock Alert), UC019 (Apply Discounts)
Actors	Store Manager

Customer Management Use Cases

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC011	Register New Customer
Overview	Create new customer profiles with personal information and membership type
Related use cases	UC014 (Manage Loyalty Points), UC015 (Upgrade Customer Membership)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC012	Update Customer Information
Overview	Modify existing customer details including contact information and preferences
Related use cases	UC013 (Search Customer Records)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC013	Search Customer Records
Overview	Locate customer information using ID, email, phone number, or name
Related use cases	UC018 (Process Sale Transaction)
Actors	Store Manager, Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC014	Manage Loyalty Points
Overview	Track, award, and redeem customer loyalty points during transactions
Related use cases	UC018 (Process Sale Transaction), UC023 (Handle Loyalty Points Redemption)
Actors	Store Manager, Customer (indirectly)

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC015	Upgrade Customer Membership
Overview	Automatically or manually upgrade customer membership tiers based on spending thresholds
Related use cases	UC014 (Manage Loyalty Points), UC016 (View Customer Analytics)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC016	View Customer Analytics

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
Overview	Analyze customer behavior, spending patterns, and membership distribution
Related use cases	UC015 (Upgrade Customer Membership), UC017 (Generate Customer Reports)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC017	Generate Customer Reports
Overview	Create detailed reports on customer demographics, spending, and loyalty program performance
Related use cases	UC016 (View Customer Analytics), UC028 (Analyze Customer Behavior)
Actors	Store Manager

Transaction Processing Use Cases

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC018	Process Sale Transaction
Overview	Handle complete sales process including item scanning, pricing, and payment
Related use cases	UC019 (Apply Discounts), UC020 (Process Payment), UC021 (Print Receipt)
Actors	Cashier, Customer (participating)

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC019	Apply Discounts
Overview	Calculate and apply various discount types including customer-based and promotional discounts
Related use cases	UC018 (Process Sale Transaction)
Actors	Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC020	Process Payment
Overview	Handle multiple payment methods including cash, cards, and mobile payments
Related use cases	UC018 (Process Sale Transaction), UC021 (Print Receipt)
Actors	Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC021	Print Receipt
Overview	Generate detailed transaction receipts with itemized purchases and totals
Related use cases	UC020 (Process Payment)
Actors	Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC022	Process Refund
Overview	Handle product returns with full or partial refunds and inventory restoration
Related use cases	UC005 (Update Stock Levels), UC024 (View Transaction History)
Actors	Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC023	Handle Loyalty Points Redemption
Overview	Process customer loyalty point usage as payment method during transactions
Related use cases	UC014 (Manage Loyalty Points), UC018 (Process Sale Transaction)
Actors	Cashier

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC024	View Transaction History

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
Overview	Browse and search historical transaction records with filtering options
Related use cases	UC022 (Process Refund), UC025 (Generate Sales Report)
Actors	Store Manager, Cashier

Reporting and Analytics Use Cases

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC025	Generate Sales Report
Overview	Create comprehensive sales analytics including revenue, transaction counts, and trends
Related use cases	UC024 (View Transaction History), UC026 (Generate Financial Summary)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC026	Generate Financial Summary
Overview	Produce overall financial performance reports including profits and expenses
Related use cases	UC025 (Generate Sales Report), UC027 (Generate Inventory Valuation Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC027	Generate Inventory Valuation Report
Overview	Calculate total inventory value, costs, and potential profits
Related use cases	UC007 (View Inventory Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC028	Analyze Customer Behavior
Overview	Study customer purchase patterns, preferences, and spending habits
Related use cases	UC016 (View Customer Analytics), UC017 (Generate Customer Reports)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC029	Track Profit Margins
Overview	Monitor and analyze profit margins by product, category, and time period
Related use cases	UC026 (Generate Financial Summary), UC027 (Generate Inventory Valuation Report)
Actors	Store Manager

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC030	Monitor Stock Movement
Overview	Track inventory turnover rates and identify fast/slow-moving products
Related use cases	UC007 (View Inventory Report), UC029 (Track Profit Margins)
Actors	Store Manager

System Administration Use Cases

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC031	Configure System Settings
Overview	Modify system-wide parameters including tax rates, business rules, and operational settings
Related use cases	UC035 (Set Tax Rates)
Actors	System Administrator

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC032	Manage User Accounts
Overview	Create, modify, and deactivate user accounts with appropriate access permissions
Related use cases	UC034 (System Maintenance)
Actors	System Administrator

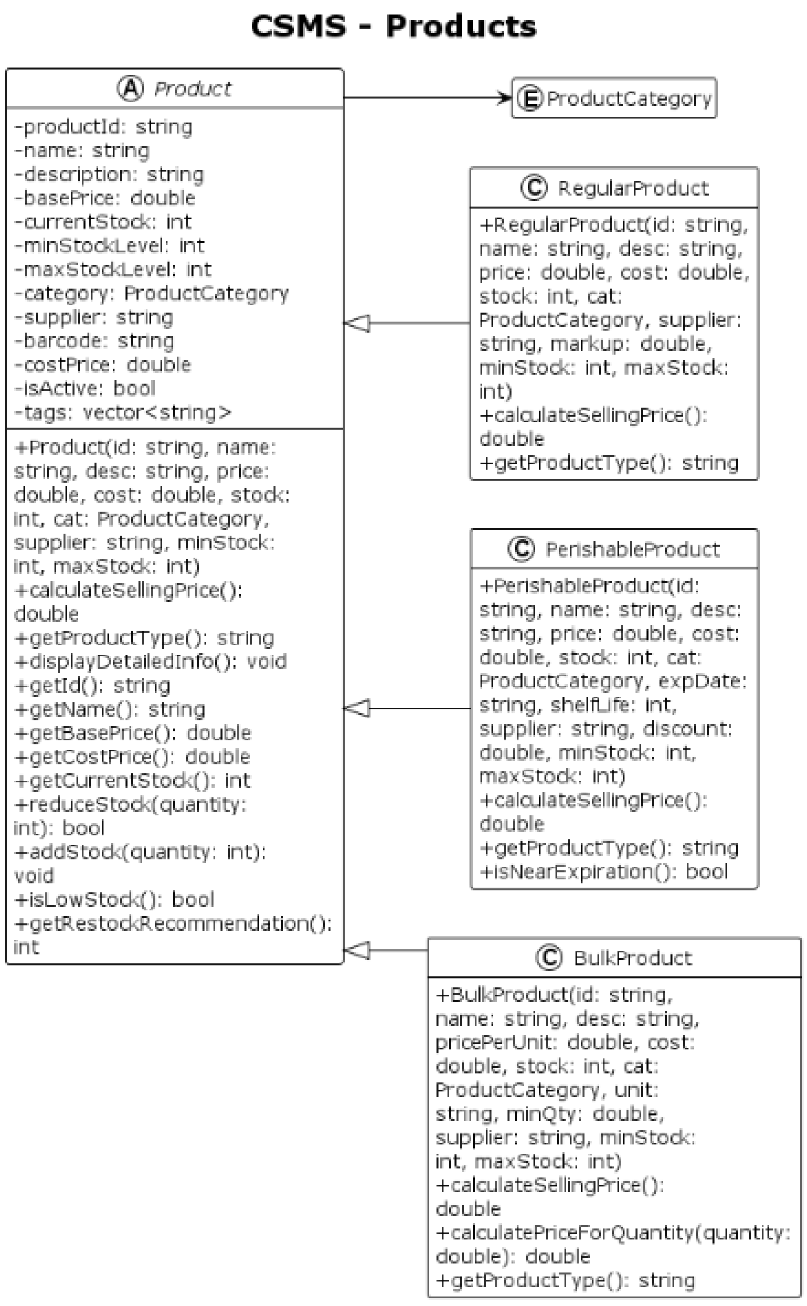
UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC033	Backup System Data
Overview	Perform automated and manual backups of system data with recovery options
Related use cases	UC034 (System Maintenance)
Actors	System Administrator

UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC034	System Maintenance
Overview	Conduct regular system maintenance including updates, optimization, and troubleshooting
Related use cases	UC032 (Manage User Accounts), UC033 (Backup System Data)
Actors	System Administrator

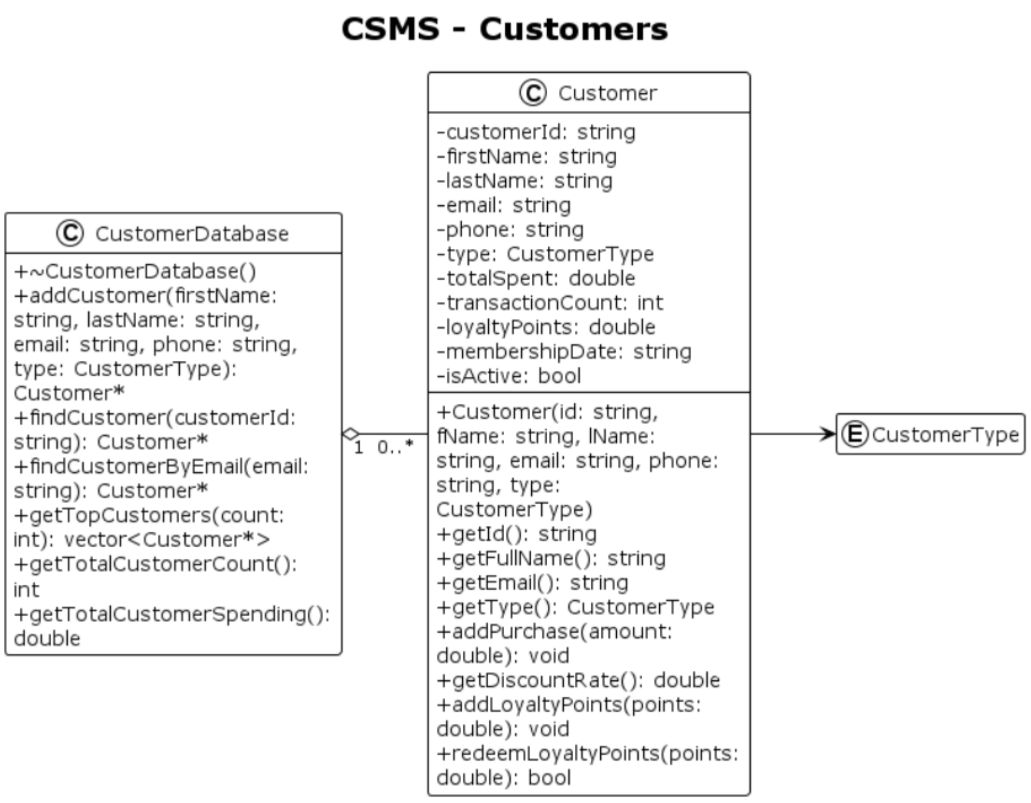
UC REFERENCE NAME/NUMBER	USECASE NAME OR NUMBER
UC035	Set Tax Rates
Overview	Configure and update tax rates and taxation rules for different product categories
Related use cases	UC031 (Configure System Settings)
Actors	System Administrator

4.2 Class Diagram

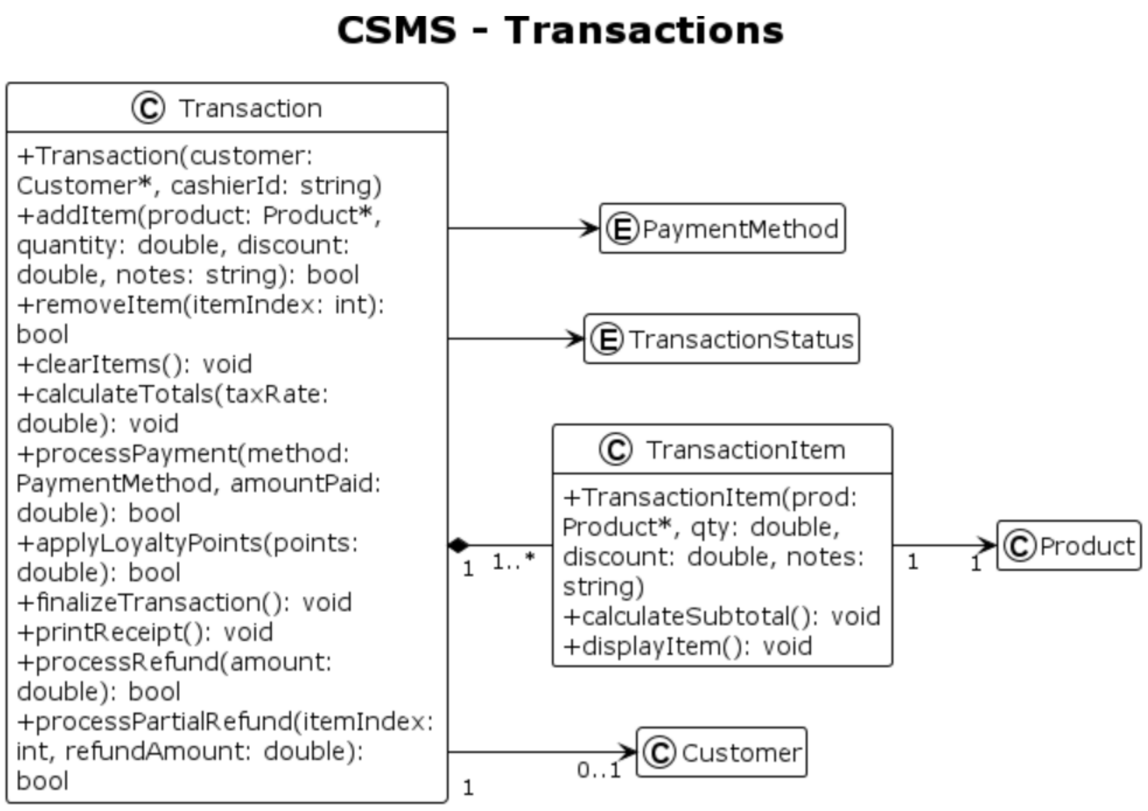
Product


CSMS - Products
A Product
-productId: string
-name: string
-description: string
-basePrice: double
-currentStock: int
-minStockLevel: int
-maxStockLevel: int
-category: ProductCategory
-supplier: string
-barcode: string
-costPrice: double
-isActive: bool
-tags: vector<string>
+Product(id: string, name: string, desc: string, price: double, cost: double, stock: int, cat: ProductCategory, supplier: string, minStock: int, maxStock: int)
+calculateSellingPrice(): double
+getProductType(): string
+displayDetailedInfo(): void
+getId(): string
+getName(): string
+getBasePrice(): double
+getCostPrice(): double
+getCurrentStock(): int
+reduceStock(quantity: int): bool
+addStock(quantity: int): void
+isLowStock(): bool
+getRestockRecommendation(): int
E ProductCategory
C RegularProduct
+RegularProduct(id: string, name: string, desc: string, price: double, cost: double, stock: int, cat: ProductCategory, supplier: string, markup: double, minStock: int, maxStock: int)
+calculateSellingPrice(): double
+getProductType(): string
C PerishableProduct
+PerishableProduct(id: string, name: string, desc: string, price: double, cost: double, stock: int, cat: ProductCategory, expDate: string, shelfLife: int, supplier: string, discount: double, minStock: int, maxStock: int)
+calculateSellingPrice(): double
+getProductType(): string
+isNearExpiration(): bool
C BulkProduct
+BulkProduct(id: string, name: string, desc: string, pricePerUnit: double, cost: double, stock: int, cat: ProductCategory, unit: string, minQty: double, supplier: string, minStock: int, maxStock: int)
+calculateSellingPrice(): double
+calculatePriceForQuantity(quantity: double): double
+getProductType(): string

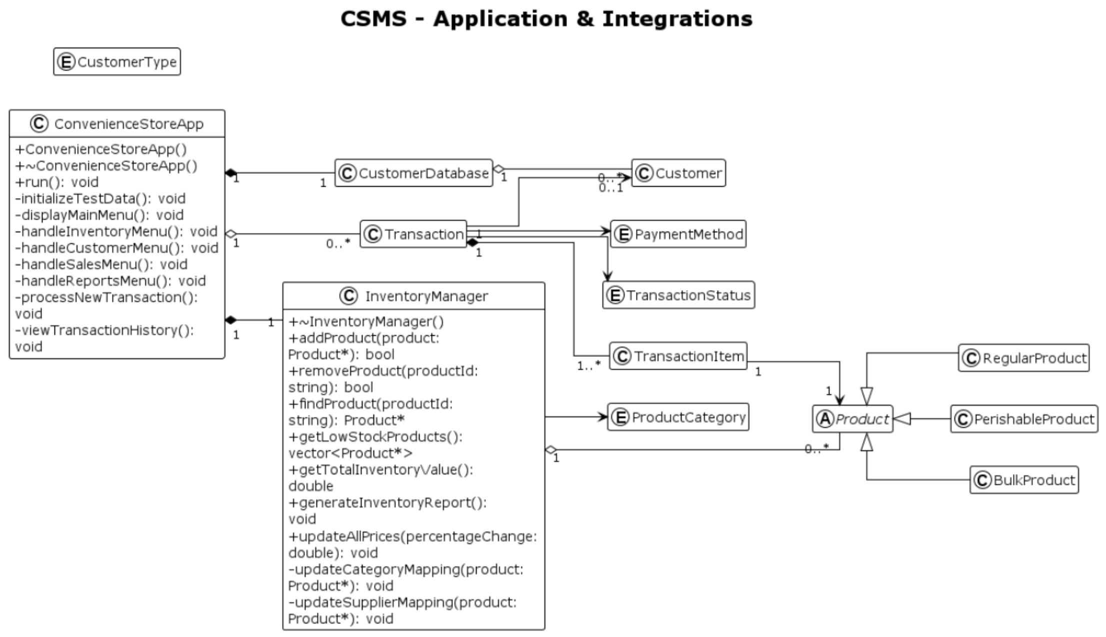
Customer & CustomerDatabase


CSMS - Customers
CustomerDatabase
+~CustomerDatabase()
+addCustomer(firstName: string, lastName: string, email: string, phone: string, type: CustomerType): Customer*
+findCustomer(customerId: string): Customer*
+findCustomerByEmail(email: string): Customer*
+getTopCustomers(count: int): vector<Customer*>
+getTotalCustomerCount(): int
+getTotalCustomerSpending(): double
1
0..*
Customer
-customerId: string
-firstName: string
-lastName: string
-email: string
-phone: string
-type: CustomerType
-totalSpent: double
-transactionCount: int
-loyaltyPoints: double
-membershipDate: string
-isActive: bool
+Customer(id: string, fName: string, lName: string, email: string, phone: string, type: CustomerType)
+getId(): string
+getFullName(): string
+getEmail(): string
+getType(): CustomerType
+addPurchase(amount: double): void
+getDiscountRate(): double
+addLoyaltyPoints(points: double): void
+redeemLoyaltyPoints(points: double): bool
CustomerType

Transaction


CSMS - Transactions
Transaction
+Transaction(customer: Customer*, cashierId: string)
+addItem(product: Product*, quantity: double, discount: double, notes: string): bool
+removeItem(itemIndex: int): bool
+clearItems(): void
+calculateTotals(taxRate: double): void
+processPayment(method: PaymentMethod, amountPaid: double): bool
+applyLoyaltyPoints(points: double): bool
+finalizeTransaction(): void
+printReceipt(): void
+processRefund(amount: double): bool
+processPartialRefund(itemIndex: int, refundAmount: double): bool
PaymentMethod
TransactionStatus
TransactionItem
+TransactionItem(prod: Product*, qty: double, discount: double, notes: string)
+calculateSubtotal(): void
+displayItem(): void
1
1..*
1
1
Product
Customer
1
0..1

Application & Integrations


CSMS - Application & Integrations
CustomerType
ConvenienceStoreApp
+ConvenienceStoreApp()
+~ConvenienceStoreApp()
+run(): void
-initializeTestData(): void
-displayMainMenu(): void
-handleInventoryMenu(): void
-handleCustomerMenu(): void
-handleSalesMenu(): void
-handleReportsMenu(): void
-processNewTransaction(): void
-viewTransactionHistory(): void
CustomerDatabase
Customer
Transaction
PaymentMethod
TransactionStatus
InventoryManager
+~InventoryManager()
+addProduct(product: Product*): bool
+removeProduct(productId: string): bool
+findProduct(productId: string): Product*
+getLowStockProducts(): vector<Product*>
+getTotalInventoryValue(): double
+generateInventoryReport(): void
+updateAllPrices(percentageChange: double): void
-updateCategoryMapping(product: Product*): void
-updateSupplierMapping(product: Product*): void
TransactionItem
ProductCategory
Product
RegularProduct
PerishableProduct
BulkProduct
1
0..*
0..1
1..*

4.3 Sequence Diagrams

Five comprehensive sequence diagrams illustrate complex business processes with **sophisticated message flows** and **proper object lifecycle management**:

SD-01: Process Sale Transaction

SD-02: Add New Product

SD-03: Customer Loyalty Management

SD-04: Inventory Low Stock Alert

SD-05: Process Refund

---

5. Object-Oriented Design Implementation

5.1 Inheritance and Polymorphism

The system demonstrates sophisticated inheritance design through the Product class hierarchy, showcasing proper abstract class usage and polymorphic behavior:

Abstract Base Class Design:

```cpp
class Product {
    // Common attributes for all products
    protected:
        string productId, name, description;
        double basePrice, costPrice;
        int currentStock, minStockLevel;
        ProductCategory category;

    public:
        // Pure virtual methods enforcing contract
        virtual double calculateSellingPrice() const = 0;
        virtual string getProductType() const = 0;

        // Virtual method with default implementation
        virtual void displayDetailedInfo() const;

        // Template method pattern
        double calculateProfitMargin() const {
            return ((calculateSellingPrice() - costPrice) / costPrice) * 100;
        }
};
```

Polymorphic Implementations:

RegularProduct implements standard markup-based pricing:

```
double RegularProduct::calculateSellingPrice() const {
    return costPrice * (1.0 + markupPercentage);
}
```

PerishableProduct implements dynamic expiration-based pricing:

```
double PerishableProduct::calculateSellingPrice() const {
    double price = basePrice;
    if (isNearExpiration()) {
        price *= (1.0 - discountRate);  // Automatic discount
    }
    return price;
}
```

BulkProduct implements unit-based pricing:

```
double BulkProduct::calculateSellingPrice() const {
    return pricePerUnit;  // Price per unit (kg, lbs, etc.)
}
```

5.2 Encapsulation and Data Hiding

The system implements comprehensive encapsulation through:

Access Control Patterns:

- **Private Attributes:** All data members are private with controlled access
- **Public Interface:** Clean, minimal public methods for external interaction
- **Protected Members:** Shared functionality in inheritance hierarchies
- **Friend Functions:** None used - maintaining strict encapsulation

Data Validation Examples:

```cpp
bool Product::reduceStock(int quantity) {
    if (currentStock >= quantity && quantity > 0) {
        currentStock -= quantity;
        return true;
    }
    return false;  // Validation failed
}

void Customer::addLoyaltyPoints(double points) {
    if (points > 0) {  // Prevent negative point addition
        loyaltyPoints += points;
    }
}
```

5.3 Composition and Aggregation

The system demonstrates both composition and aggregation relationships:

Composition Relationships (Strong Ownership):

```cpp
class Transaction {
private:
    vector<TransactionItem> items;  // Composition: items owned by transaction

public:
    ~Transaction() {
        // TransactionItems automatically destroyed with Transaction
    }
};
```

Aggregation Relationships (Weak Ownership):

```cpp
class InventoryManager {
private:
    map<string, Product*> products;  // Aggregation: products can exist independently

public:
    ~InventoryManager() {
        for (auto& pair : products) {
            delete pair.second;  // Cleanup responsibility
        }
    }
};
```

5.4 Method Overriding and Virtual Functions

Strategic use of virtual functions enables runtime polymorphism:

```cpp
// Base class virtual method
virtual void Product::displayDetailedInfo() const {
    // Common display logic
}

// Derived class override with extended functionality
void PerishableProduct::displayDetailedInfo() const override {
    Product::displayDetailedInfo();  // Call base implementation
    cout << "Expiration Date: " << expirationDate << endl;
    if (isNearExpiration()) {
        cout << "🔥 NEAR EXPIRATION! Discount applied" << endl;
    }
}
```

---

6. SOLID Principles Application

The Convenience Store Management System rigorously applies all five SOLID principles, demonstrating enterprise-level software design practices:

6.1 Single Responsibility Principle (SRP)

Each class has a **single, well-defined responsibility** with high cohesion:

Examples of SRP Implementation:

- `Product`: Manages product data and behavior only
- `Customer`: Handles customer information and loyalty logic exclusively
- `Transaction`: Processes sales transactions without inventory or customer management
- `InventoryManager`: Manages product collections and inventory operations only
- `CustomerDatabase`: Focuses solely on customer data management

Responsibility Separation:

```
class Transaction {
    // ONLY transaction-related responsibilities:
    // - Calculate totals and taxes
    // - Process payments
    // - Generate receipts
    // - Manage transaction state

    // NOT responsible for:
    // - Inventory management (delegated to InventoryManager)
    // - Customer data updates (delegated to Customer class)
    // - Product pricing (delegated to Product classes)
};
```

6.2 Open/Closed Principle (OCP)

The system is **open for extension, closed for modification**:

Extension Points:

1. **New Product Types:** Add new classes inheriting from Product without modifying existing code
2. **Payment Methods:** Extend PaymentMethod enum without changing Transaction logic

3. **Customer Types:** Add new tiers without modifying existing customer logic
4. **Report Types:** Add new reporting methods without changing core business logic

Example - Adding New Product Type:

```cpp
// New product type can be added without modifying existing code
class DigitalProduct : public Product {
private:
    string downloadUrl;
    int downloadLimit;

public:
    double calculateSellingPrice() const override {
        // Digital product pricing logic
        return basePrice;  // No inventory costs
    }

    string getProductType() const override {
        return "Digital";
    }
};
```

6.3 Liskov Substitution Principle (LSP)

Derived classes can replace base classes without breaking functionality:

Substitution Examples:

```cpp
// Any Product* can be substituted with derived class instances
void processTransaction(Product* product) {
    double price = product->calculateSellingPrice();  // Works for all product types
    string type = product->getProductType();          // Polymorphic behavior
    product->displayDetailedInfo();                   // Virtual method dispatch
}

// LSP compliance - all substitutions work correctly:
Product* products[] = {
    new RegularProduct(...),
    new PerishableProduct(...),
    new BulkProduct(...)
};
```

Contract Preservation:

- All derived classes properly implement abstract methods
- Preconditions are not strengthened in derived classes
- Postconditions are not weakened in derived classes
- Invariants are maintained across the inheritance hierarchy

6.4 Interface Segregation Principle (ISP)

Classes depend only on methods they actually use:

Minimal Interface Design:

```cpp
// Product interface is minimal and focused
class Product {
public:
    // Core interface - only essential methods
    virtual double calculateSellingPrice() const = 0;
    virtual string getProductType() const = 0;

    // Clients only depend on methods they need
    string getId() const;            // For identification
    int getCurrentStock() const;     // For inventory checks
    bool reduceStock(int qty);      // For transactions
};

// Specialized interfaces for specific needs
class PerishableProduct : public Product {
public:
    // Extended interface only for clients needing expiration logic
    bool isNearExpiration() const;
    int getDaysUntilExpiration() const;
};
```

6.5 Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not concretions:

Abstraction Dependencies:

```
class Transaction {
private:
    Customer* customer;           // Depends on Customer abstraction
    vector<TransactionItem> items; // Contains abstractions

public:
    bool addItem(Product* product, double quantity) {
        // Depends on Product abstraction, not concrete types
        // Works with RegularProduct, PerishableProduct, BulkProduct
        if (product->getCurrentStock() >= quantity) {
            items.push_back(TransactionItem(product, quantity));
            return true;
        }
        return false;
    }
};
```

Dependency Injection Patterns:

```cpp
class ConvenienceStoreApp {
private:
    InventoryManager inventory;      // Composition with concrete classes
    CustomerDatabase customerDB;     // But interface-based interactions

public:
    ConvenienceStoreApp() {
        // Dependencies injected through constructor/initialization
        initializeTestData();
    }

    void processTransaction() {
        // High-level policy depends on abstractions
        Product* product = inventory.findProduct(productId);  // Returns Product*
        Customer* customer = customerDB.findCustomer(customerId);  // Returns Customer*
    }
};
```

---

7. Design Patterns and Best Practices

7.1 Design Patterns Implementation

The system incorporates multiple **Gang of Four design patterns** and enterprise patterns:

7.1.1 Strategy Pattern

Implementation: Product pricing strategies **Purpose:** Encapsulate different pricing algorithms

```cpp
// Context: Product class defines strategy interface
```

```cpp
class Product {
public:
    virtual double calculateSellingPrice() const = 0;  // Strategy method
};

// Concrete Strategies:
class RegularProduct : public Product {
    double calculateSellingPrice() const override {
        return costPrice * (1.0 + markupPercentage);  // Markup strategy
    }
};

class PerishableProduct : public Product {
    double calculateSellingPrice() const override {
        double price = basePrice;
        if (isNearExpiration()) {
            price *= (1.0 - discountRate);  // Discount strategy
        }
        return price;
    }
};

class BulkProduct : public Product {
    double calculateSellingPrice() const override {
        return pricePerUnit;  // Unit-based strategy
    }
};
```

7.1.2 Factory Pattern (Implicit)

Implementation: Product creation through inheritance **Purpose:** Encapsulate object creation logic

```cpp
// Factory-like creation in ConvenienceStoreApp
Product* createProduct(int productType, /* other parameters */) {
    switch (productType) {
        case 1: return new RegularProduct(/* params */);
        case 2: return new PerishableProduct(/* params */);
        case 3: return new BulkProduct(/* params */);
        default: return nullptr;
    }
}
```

7.1.3 Composite Pattern

Implementation: Transaction and TransactionItem relationship **Purpose:** Treat individual objects and compositions uniformly

```cpp
class Transaction {
private:
    vector<TransactionItem> items;  // Composite structure

public:
    double calculateTotal() const {
        double total = 0.0;
        for (const auto& item : items) {  // Iterate over composite elements
            total += item.getSubtotal();
        }
        return total;
    }

    void displayReceipt() const {
        for (const auto& item : items) {  // Uniform treatment
            item.displayItem();           // Each item handles its own display
        }
    }
};
```

7.1.4 Observer Pattern (Conceptual)

Implementation: Low stock alert system **Purpose:** Notify interested parties of state changes

```cpp
class Product {
public:
    bool isLowStock() const {
        return currentStock <= minStockLevel;
    }

    int getRestockRecommendation() const {
        if (isLowStock()) {
            return maxStockLevel - currentStock;  // Trigger for observers
        }
        return 0;
    }
};

class InventoryManager {
public:
    vector<Product*> getLowStockProducts() const {
        vector<Product*> lowStockItems;
        for (const auto& pair : products) {
            if (pair.second->isLowStock()) {  // Observer pattern trigger
                lowStockItems.push_back(pair.second);
            }
        }
        return lowStockItems;
    }
};
```

7.2 Enterprise Patterns

7.2.1 Domain Model Pattern

Rich objects with business behavior embedded in domain entities:

```cpp
class Customer {
```

```
public:
    void addPurchase(double amount) {
        totalSpent += amount;
        transactionCount++;

        // Business logic embedded in domain object
        double pointsMultiplier = getPointsMultiplier();
        addLoyaltyPoints(amount * 0.01 * pointsMultiplier);

        // Check for automatic upgrade
        if (isEligibleForUpgrade()) {
            upgradeCustomerType();
        }
    }

private:
    double getPointsMultiplier() const {
        switch (type) {
            case CustomerType::PREMIUM: return 1.5;
            case CustomerType::VIP: return 2.0;
            case CustomerType::EMPLOYEE: return 3.0;
            default: return 1.0;
        }
    }
};
```

7.2.2 Service Layer Pattern

Business services coordinate domain objects:

```
class InventoryManager {
public:
    void generateLowStockReport() const {
        auto lowStockProducts = getLowStockProducts();     // Service coordination
        auto outOfStockProducts = getOutOfStockProducts(); // Multiple domain queries

        // Business logic coordination
        if (!outOfStockProducts.empty()) {
            displayCriticalAlerts(outOfStockProducts);
        }
```

```cpp
        if (!lowStockProducts.empty()) {
            displayWarningAlerts(lowStockProducts);
        }
    }
};
```

7.3 Memory Management Best Practices

7.3.1 RAII (Resource Acquisition Is Initialization)

```cpp
class InventoryManager {
private:
    map<string, Product*> products;

public:
    ~InventoryManager() {
        // RAII: Automatic cleanup in destructor
        for (auto& pair : products) {
            delete pair.second;
        }
    }

    bool removeProduct(const string& productId) {
        auto it = products.find(productId);
        if (it != products.end()) {
            delete it->second;    // Explicit cleanup
            products.erase(it);
            return true;
        }
        return false;
    }
};
```

7.3.2 Exception Safety

```cpp
bool Transaction::addItem(Product* product, double quantity,
double discount) {
    // Strong exception safety guarantee
    if (!product || !product->getIsActive() || quantity <= 0) {
        return false;  // Early validation
    }

    if (product->getCurrentStock() < static_cast<int>(ceil(quantity))) {
        return false;  // State unchanged on failure
    }

    // Only modify state after all validations pass
    items.push_back(TransactionItem(product, quantity, discount));
    return true;
}
```

7.4 Performance Optimizations

7.4.1 Efficient Data Structures

```cpp
class InventoryManager {
private:
    map<string, Product*> products;                    // O(log n) lookups
    map<ProductCategory, vector<Product*>> productsByCategory; // Category indexing
    map<string, vector<Product*>> productsBySupplier;     // Supplier indexing

public:
    Product* findProduct(const string& productId) {
        auto it = products.find(productId);               // O(log n) complexity
        return (it != products.end()) ? it->second : nullptr;
    }
};
```

7.4.2 Lazy Evaluation

```cpp
class Customer {
public:
    bool isEligibleForUpgrade() const {
        // Lazy evaluation - only compute when needed
        if (type == CustomerType::REGULAR && totalSpent >= 500.0) {
            return true;
        }
        if (type == CustomerType::PREMIUM && totalSpent >= 2000.0) {
            return true;
        }
        return false;
    }
};
```