Store Management System - Technical Documentation

Date: July 19, 2025
Authors: Tomodan Zeno-Tudor & Suci Ianis Luca
Group: 3 @ West University of Timisoara, Faculty of Mathematics and Computer Science A.I.
Language: C++

Table of Contents

System Overview

The Store Management System is a console-based C++ application designed to facilitate store operations through a dual-user approach: administrators who manage inventory and customers who interact with the store through shopping carts and orders.

Key Objectives:

- **Inventory Management**: Complete CRUD operations for product management
- **Order Processing**: Streamlined cart-to-order workflow with real-time stock updates
- **User Authentication**: Role-based access control with predefined user accounts
- **Data Persistence**: File-based storage system using text files
- **Real-time Stock Management**: Automatic stock updates during transactions
- **Input Validation**: Comprehensive validation for security and data integrity

System Requirements:

- **Operating System**: Windows, Linux, macOS
- **Compiler**: G++ with C++11 support or higher
- **Memory**: Minimum 512MB RAM
- **Storage**: 50MB for application and data files

Architecture Design

System Architecture Pattern: Layered Architecture

```
┌─────────────────────────────────────┐
│    Presentation Layer       │ ← Console UI & Menu System
├─────────────────────────────────────┤
│    Business Logic Layer      │ ← Core System Operations
├─────────────────────────────────────┤
│    Data Access Layer        │ ← File I/O Operations
├─────────────────────────────────────┤
│    Data Storage Layer       │ ← Text Files (stoc.txt, etc.)
└─────────────────────────────────────┘
```

Design Principles Applied:

- **Single Responsibility Principle**: Each class has one primary responsibility
- **Open/Closed Principle**: System is extensible without modifying existing code
- **Encapsulation**: Data hiding through public member variables and clear interfaces
- **Modularity**: Clear separation of concerns across different components

Class Structure

1. Date Class

cpp

```cpp
class Date {
public:
    int day, month, year;

    Date();
    Date(int d, int m, int y);
    string toString() const;
    static Date getCurrentDate();
};
```

Responsibilities:

- Date representation and formatting
- Current date generation (hardcoded to 24/5/2025)
- Date string conversion

Key Methods:

- `toString()`: Formats date as "DD/MM/YYYY"
- `getCurrentDate()`: Returns current system date (24/5/2025)

2. Product Class

```cpp
class Product {
public:
    string barcode;
    string name;
    int quantity;
    double price;

    Product();
    Product(string bc, string n, int q, double p);
    string toString() const;
};
```

Responsibilities:

- Product data encapsulation with public member access
- Product information formatting for file storage
- Product data representation

Data Structure:

- All members are public for direct access
- No private validation - handled at system level

3. Order Class

```cpp
```

```cpp
class Order {
public:
    vector<Product> products;
    Date date;

    Order();
    Order(vector<Product> prods, Date d);
};
```

Responsibilities:

- Order composition management
- Product aggregation for completed purchases
- Order date tracking with automatic date assignment

4. User Class

```cpp
class User {
public:
    string username;
    string password;
    bool isAdmin;

    User(string u, string p, bool admin);
};
```

Responsibilities:

- User credential storage
- Role-based access control flag
- Authentication state tracking

5. StoreSystem Class (Main Controller)

```cpp
```

```cpp
class StoreSystem {
private:
    vector<Product> stock;
    vector<Order> orders;
    vector<Product> cart;
    vector<User> users;
    User* currentUser;

    // File constants
    const string STOCK_FILE = "stoc.txt";
    const string ORDERS_FILE = "comenzi.txt";
    const string CART_FILE = "cos_cumparaturi.txt";
    const string USERS_FILE = "users.txt";

    // Color constants
    const string RED = "\033[31m";
    const string RESET = "\033[0m";

public:
    // System initialization
    StoreSystem();
    void initializeUsers();

    // Validation and utilities
    bool isValidInput(const string& input);
    void clearScreen();

    // Authentication methods
    bool login();
    void logout();

    // File operations
    void loadStock();
    void saveStock();
    void loadOrders();
    void saveOrders();
    void loadCart();
    void saveCart();

    // Admin methods
    void viewStockProducts();
    void addProduct(string, string, int, double);
    void deleteProduct(string);
    void modifyProduct(string, string, double);
    void viewOrders();
    // Customer methods
    void viewCart();
    void addToCart(string, int);
    void modifyCartProduct(string, int);
    void deleteFromCart(string);
    void purchase();

    // System methods
    void run();
    void showMenu();
    void handleMenu();
};
```

File System Management

File Structure Organization:

1. stoc.txt (Inventory File)

```
Format:
[Number of products]
[Barcode] [Name] [Quantity] [Price]
...

Example:
5
001 Laptop_HP 10 899.99
002 Mouse_Logitech 50 25.99
003 Keyboard_Mechanical 30 129.99
004 Monitor_Samsung 8 299.99
005 Headphones_Sony 20 149.99
```

2. comenzi.txt (Orders File)

```
Format:
[Date]
[Barcode1] [Barcode2] [Barcode3] ...

Example:
24/5/2025
001 002 003
24/5/2025
002 002
```

3. cos_cumparaturi.txt (Shopping Cart File)

```
Format:
[Barcode] [Quantity]
...

Example:
001 2
003 1
```

File I/O Operations:

Loading Data:

```cpp
void loadStock() {
    ifstream file(STOCK_FILE);
    if (!file.is_open()) return;

    int numProducts;
    file >> numProducts;
    file.ignore();

    stock.clear();
    for (int i = 0; i < numProducts; i++) {
        string line;
        getline(file, line);
        if (!line.empty()) {
            istringstream iss(line);
            string barcode, name;
            int quantity;
            double price;

            iss >> barcode >> name >> quantity >> price;
            stock.push_back(Product(barcode, name, quantity, price));
        }
    }
    file.close();
}
```

Saving Data:

```cpp
```

```
void saveStock() {
    ofstream file(STOCK_FILE);
    if (!file.is_open()) return;

    file << stock.size() << endl;
    for (const auto& product : stock) {
        file << product.toString() << endl;
    }
    file.close();
}
```

Data Integrity Measures:

- **File Existence Checks**: All file operations check for successful file opening
- **Automatic Recovery**: System continues operation if files don't exist
- **Immediate Persistence**: Changes are saved immediately after operations
- **Data Validation**: Input validation before file operations

Authentication System

Security Architecture:

User Roles:

1. **Administrator (isAdmin = true)**
   - Full inventory management access
   - Order viewing capabilities
   - Product CRUD operations
2. **Customer (isAdmin = false)**
   - Shopping cart management
   - Order placement
   - Product browsing

Authentication Flow:

```
User Input → Input Validation → Credential Validation → Role Assignment → Session Management
```

Default User Accounts:

Username	Password	Role	Permissions
admin	admin123	Admin	Full Access
ion	ion123	Admin	Full Access
user	user123	Customer	Limited
maria	maria123	Customer	Limited

Input Validation:

```cpp
bool isValidInput(const string& input) {
    for (char c : input) {
        if (!isalpha(c) && !(c >= '1' && c <= '9')) {
            return false;
        }
    }
    return !input.empty();
}
```

Validation Rules:

- Only alphabetic characters allowed
- Only numbers 1-9 allowed (0 is excluded)
- Empty strings are rejected
- Special characters are rejected

Session Management:

- **Current User Tracking**: Pointer to active user object
- **Role Validation**: Method-level permission checks
- **Automatic Cart Loading**: Cart loaded per customer session
- **Secure Logout**: Proper session cleanup

Core Functionalities

Administrator Functions:

1. Inventory Management

Add Product:

```cpp
```

```
Algorithm:
1. Validate admin permissions
2. Check for existing barcode (prevent duplicates)
3. If duplicate found: Display error and existing product info
4. If new: Create product entry
5. Save to file system
6. Confirm operation success
```

Delete Product:

```cpp
Algorithm:
1. Verify admin access
2. Search product by barcode using remove_if
3. Remove from stock vector
4. Update persistent storage
5. Provide operation feedback
```

Modify Product:

```cpp
Algorithm:
1. Authenticate admin user
2. Locate product by barcode
3. Validate modification type (price/quantity)
4. Apply changes with type casting for quantity
5. Persist changes to file
```

2. Order Management

- **View Orders**: Chronological order listing with product details and dates
- **Order Tracking**: Complete order history with date stamps

Customer Functions:

1. Shopping Cart Management

Add to Cart:

```cpp
```

```
Algorithm:
1. Verify customer permissions
2. Find product in stock
3. Validate stock availability
4. Check for existing cart item (update quantity if exists)
5. Add new item or update existing
6. Save cart state
7. Provide user feedback
```

Cart Modification:

```cpp
Algorithm:
1. Locate cart item by barcode
2. Validate new quantity against stock
3. If quantity <= 0: Remove item from cart
4. Otherwise: Update quantity with stock validation
5. Save cart changes
```

2. Order Processing

Purchase Flow:

```cpp
Algorithm:
1. Validate non-empty cart
2. Check stock availability for all cart items
3. Validate sufficient stock for each product
4. Create order record with current date
5. Update stock quantities (subtract cart quantities)
6. Calculate and display total amount
7. Clear shopping cart
8. Save all changes to files (cart, stock, orders)
```

User Interface Design

Menu Architecture:

Admin Menu Structure:

```
=== MAIN MENU ===
1. View Stock Products
2. Add Product
3. Delete Product
4. Modify Product
5. View Orders
6. Clear Screen
0. Logout
```

Customer Menu Structure:

```
=== MAIN MENU ===
1. View Cart
2. Add Product to Cart
3. Modify Cart Product
4. Delete from Cart
5. Purchase
6. View Available Products
7. Clear Screen
0. Logout
```

Interface Guidelines:

- **Consistent Formatting**: Uniform table layouts using `setw()` for alignment
- **Clear Navigation**: Numbered menu options with logical flow
- **Input Validation**: Real-time feedback for invalid entries
- **Color Coding**: Red text for errors using ANSI escape codes
- **Cross-Platform**: Clear screen functionality for Windows/Unix systems

Display Formatting:

```cpp
// Product table display
cout << left << setw(12) << "Barcode" << setw(20) << "Name"
     << setw(10) << "Quantity" << setw(10) << "Price" << endl;
cout << string(52, '-') << endl;
```

Data Flow Diagrams

Admin Product Management Flow:

```
Admin Login → Menu Selection → Product Operation → Data Validation →
Stock Vector Update → File Persistence → User Confirmation
```

Customer Purchase Flow:

```
Customer Login → Browse Products → Add to Cart → Review Cart →
Purchase Validation → Stock Update → Order Creation → Cart Cleanup
```

System Startup Flow:

```
Application Start → Initialize Users → Load Stock Data →
Load Orders → Authentication Prompt → Role-Based Menu Display
```

Error Handling

Error Categories:

1. File System Errors

- **File Access Issues**: Graceful handling when files can't be opened
- **Missing Files**: System continues with empty data structures
- **File Format Issues**: Basic parsing with line-by-line processing

Handling Strategy:

```cpp
ifstream file(STOCK_FILE);
if (!file.is_open()) return; // Graceful failure
```

2. Data Validation Errors

- **Invalid Input Characters**: Alphanumeric validation with specific rules
- **Duplicate Barcodes**: Prevention of duplicate product entries
- **Negative Quantities**: Implicit validation through unsigned operations
- **Empty Cart Operations**: Purchase prevention with empty cart

3. Business Logic Errors

- **Insufficient Stock**: Stock validation before purchase
- **Product Not Found**: Comprehensive search validation
- **Access Control**: Role-based operation restrictions

4. Authentication Errors

- **Invalid Credentials**: Username/password mismatch handling
- **Access Denied**: Role-based permission violations
- **Input Format**: Character validation for security

Error Recovery Mechanisms:

- **Graceful Degradation**: System continues operation despite errors
- **User Guidance**: Clear error messages with context
- **State Preservation**: Cart and data persistence during errors
- **Retry Capability**: Users can re-attempt operations

Installation & Setup

Compilation Instructions:

Standard Compilation:

```bash
g++ -std=c++11 -o store_system paste.txt
```

With Debug Information:

```bash
g++ -std=c++11 -g -O0 -o store_system paste.txt
```

Optimized Build:

```bash
g++ -std=c++11 -O2 -o store_system paste.txt
```

Initial Setup:

1. File Preparation:

The system will automatically handle missing files, but you can pre-create:

- `stoc.txt` (will be created on first product addition)
- `comenzi.txt` (will be created on first order)
- `cos_cumparaturi.txt` (will be created when customers add to cart)

2. Permissions:

Ensure read/write permissions for data files:

```bash
chmod 644 *.txt
```

3. First Run:

```bash
./store_system
```

The system will display default credentials:

- Admin: username='admin', password='admin123'
- Admin: username='ion', password='ion123'
- Customer: username='user', password='user123'
- Customer: username='maria', password='maria123'

Sample Data Setup:

Initial Stock (stoc.txt):

```
5
001 Laptop_HP 10 899.99
002 Mouse_Logitech 50 25.99
003 Keyboard_Mechanical 30 129.99
004 Monitor_Samsung 8 299.99
005 Headphones_Sony 20 149.99
```

API Reference

Core Methods Documentation:

Authentication Methods:

`bool login()`

- **Purpose**: Authenticate user credentials with input validation
- **Parameters**: None (interactive input)
- **Returns**: Boolean success status
- **Side Effects**: Sets currentUser pointer, loads customer cart

- **Validation**: Alphanumeric input validation

`void logout()`

- **Purpose**: Terminate user session
- **Parameters**: None
- **Returns**: Void
- **Side Effects**: Clears currentUser, displays farewell message

Admin Methods:

`void addProduct(string barcode, string name, int quantity, double price)`

- **Purpose**: Add new product to inventory
- **Parameters**:
  - `barcode`: Unique product identifier
  - `name`: Product display name
  - `quantity`: Initial stock quantity
  - `price`: Product unit price
- **Preconditions**: Admin authentication required
- **Validations**: Duplicate barcode prevention
- **Postconditions**: Product added to stock, files updated

`void deleteProduct(string barcode)`

- **Purpose**: Remove product from inventory
- **Parameters**: `barcode` - Product identifier to remove
- **Algorithm**: Uses `remove_if` with lambda function
- **Postconditions**: Product removed, stock file updated

`void modifyProduct(string type, string barcode, double newValue)`

- **Purpose**: Update product attributes
- **Parameters**:
  - `type`: "price" or "quantity"
  - `barcode`: Product identifier
  - `newValue`: New value to set
- **Validations**: Type validation, product existence check

Customer Methods:

`void addToCart(string barcode, int quantity)`

- **Purpose**: Add products to shopping cart
- **Parameters**:
  - `barcode`: Product identifier
  - `quantity`: Desired quantity
- **Validations**: Stock availability, product existence
- **Side Effects**: Cart updated, files saved

`void purchase()`

- **Purpose**: Convert cart to order
- **Parameters**: None
- **Preconditions**: Non-empty cart, sufficient stock
- **Algorithm**: Multi-step validation and update process
- **Postconditions**: Order created, stock updated, cart cleared

Utility Methods:

`bool isValidInput(const string& input)`

- **Purpose**: Validate input for security
- **Rules**: Only letters and numbers 1-9 allowed
- **Returns**: Boolean validation result

`void clearScreen()`

- **Purpose**: Clear console screen
- **Platform Support**: Windows (`cls`) and Unix (`clear`)

Testing Guidelines

Testing Strategy:

1. Unit Testing

Test Product Operations:

```cpp
```

```cpp
void testProductCreation() {
    Product p("001", "TestProduct", 10, 99.99);
    assert(p.barcode == "001");
    assert(p.quantity == 10);
    assert(p.price == 99.99);
}
```

Test Input Validation:

```cpp
void testInputValidation() {
    StoreSystem store;
    assert(store.isValidInput("admin123") == true);
    assert(store.isValidInput("user@123") == false);
    assert(store.isValidInput("test0") == false);  // 0 not allowed
    assert(store.isValidInput("") == false);
}
```

2. Integration Testing

Authentication Flow Testing:

1. Test valid credentials for all user types
2. Test invalid input rejection
3. Test role-based menu access
4. Test session management

File Operations Testing:

1. Test save/load cycles for all file types
2. Test missing file handling
3. Test file permission scenarios

3. User Acceptance Testing

Complete Admin Workflow:

1. Login as admin with valid credentials
2. Add sample products with various data
3. Modify product details (price and quantity)
4. Delete products and verify removal
5. View stock and orders

6. Test duplicate barcode prevention
7. Logout and verify session cleanup

Complete Customer Workflow:

1. Login as customer
2. Browse available products
3. Add multiple items to cart
4. Modify cart contents (increase/decrease quantities)
5. Remove items from cart
6. Complete purchase and verify stock updates
7. Test insufficient stock scenarios

Test Data Sets:

Valid Test Cases:

- Standard product operations with valid data
- Normal user workflows with expected inputs
- Boundary value testing (minimum/maximum quantities, prices)
- Role-based access testing

Invalid Test Cases:

- Invalid input characters (special symbols, number 0)
- Insufficient permissions attempts
- Out-of-stock purchase attempts
- Empty cart purchase attempts
- Duplicate barcode additions

Future Enhancements

Planned Features:

1. Enhanced Security

- Password encryption using SHA-256 or similar
- Session timeout implementation
- Input sanitization improvements
- User registration system with validation

2. Advanced Inventory Management

- Product categories and subcategories
- Low stock alerts and notifications
- Bulk import/export functionality using CSV
- Inventory valuation and reporting

3. Improved User Interface

- GUI implementation using Qt or similar framework
- Web-based interface with REST API
- Mobile application support
- Real-time dashboard with live updates

4. Database Integration

- Migration from file-based to SQLite/MySQL
- SQL query support for advanced reporting
- Data backup and recovery systems
- Multi-user concurrent access with locking

5. Advanced Features

- Sales analytics and trend analysis
- Customer behavior tracking
- Revenue forecasting and reporting
- Multi-store support with centralized management

6. Code Quality Improvements

- Exception handling with try-catch blocks
- Memory management optimization
- Design pattern implementation (Observer, Factory)
- Comprehensive logging system

Technical Debt Management:

Immediate Improvements:

1. **Input Validation Enhancement**: More robust validation beyond alphanumeric
2. **Error Handling**: Comprehensive exception handling framework
3. **Code Documentation**: Detailed inline documentation

4. **Memory Management**: Smart pointer usage where applicable
5. **Configuration**: External configuration file support

Architecture Improvements:

1. **Separation of Concerns**: Better layer separation
2. **Dependency Injection**: Reduce tight coupling
3. **State Management**: Improved session and state handling
4. **Testing Framework**: Automated unit testing implementation

Conclusion

The Store Management System successfully implements a comprehensive retail management solution with role-based access control, persistent data storage, and intuitive console interface. The system demonstrates solid object-oriented principles while maintaining simplicity and functionality.

Key Achievements:

- **Role-based Access Control**: Secure multi-user environment with proper session management
- **Data Persistence**: Reliable file-based storage system with automatic recovery
- **Business Logic Implementation**: Complete inventory and order management workflow
- **User-friendly Interface**: Intuitive console-based interaction with clear navigation
- **Input Security**: Comprehensive input validation for security and data integrity
- **Error Resilience**: Graceful error handling and user feedback

System Metrics:

- **Lines of Code**: ~400 lines of functional code
- **Classes**: 5 main classes with clear responsibilities
- **Methods**: 20+ public methods covering all functionality
- **File Operations**: 6 file I/O methods with error handling
- **User Roles**: 2 distinct user types with 4 predefined accounts
- **Core Features**: 15+ implemented features covering full workflow

Current Limitations:

- File-based storage limits scalability
- No password encryption
- Limited concurrent user support
- Console-only interface
- Hardcoded configuration values

The system provides an excellent foundation for future enhancements and demonstrates practical application of software engineering principles in a retail management context.