Vending Machine Simulator - Project Report

Introduction

The Vending Machine Simulation comprises a complete object-oriented design implementing SOLID principles to model a real-world vending machine. This system simulates all aspects of a vending machine including inventory management, payment processing, transaction handling, and user interaction without requiring a graphical interface. The implementation demonstrates professional OOP practices through clean architecture, separation of concerns, and comprehensive testing.

Design and Functionality

The vending machine system is designed using a layered architecture that separates business logic, data models, and user interface concerns. The core components include drink hierarchy management, inventory control, payment processing, and transaction tracking.

The system supports:

- Multiple drink types (Soda, Juice, Water) with extensible architecture
- Real-time inventory management with stock tracking
- Cash payment processing with change calculation
- Transaction history and statistics
- Comprehensive error handling for invalid operations
- Admin mode for inventory management
- Command-line interface for user interaction

Features

Object-Oriented Design Principles

The implementation demonstrates all four OOP pillars (Encapsulation, Abstraction, Inheritance, Polymorphism) and follows SOLID principles for maintainable and extensible code.

Drink Hierarchy Management

A flexible inheritance structure allows for easy addition of new drink types without modifying existing code, following the Open/Closed principle. The abstract `Drink` base class defines the contract, while `Soda`, `Juice`, and `Water` provide specialized implementations.

Inventory Control

Real-time stock tracking with methods for restocking, reducing stock, and checking availability across all drink items. The `Inventory` class encapsulates all stock management operations.

Payment Processing

Abstract payment service layer supporting multiple payment methods with cash payment implementation. The `PaymentService` abstraction enables easy extension for credit card or mobile payments.

Transaction Management

Complete transaction history tracking with statistics for successful and failed transactions. Each transaction records item details, amounts, timestamps, and status.

Error Handling

Comprehensive exception hierarchy for handling insufficient funds, unavailable items, invalid selections, and other edge cases:

- `InsufficientFundsError` - When balance is too low
- `OutOfStockError` - When item is unavailable
- `InvalidProductError` - When product code doesn't exist
- `InvalidAmountError` - When invalid money amount is inserted

Testing Suite

87 comprehensive unit and integration tests covering all components with 100% pass rate.

Admin Mode

Password-protected administrative panel for inventory management, restocking, adding new items, and viewing detailed transaction reports.

Protocol Specification

The vending machine system uses a clean API protocol with well-defined methods for each operation. All interactions follow object-oriented patterns with proper encapsulation and type safety.

Message Format (Method Calls)

Initialize Machine:

- Method: `VendingMachine(inventory, payment_service, transaction_service)`
- Parameters: Inventory instance, optional PaymentService, optional TransactionService
- Returns: VendingMachine instance

Insert Money:

- Method: `insert_money(amount)`
- Parameters: `amount` (float) - money to insert
- Returns: Current balance (float)
- Raises: `InvalidAmountError` if amount <= 0

Select Item:

- Method: `select_item(code)`
- Parameters: `code` (str) - item code (e.g., "A1")
- Returns: Tuple of (success: bool, message: str, transaction: dict)

Refund:

- Method: `refund()`
- Parameters: None
- Returns: Refund amount (float)

Get Menu:

- Method: `get_menu()`
- Parameters: None
- Returns: Dictionary of all items with name, price, quantity, category, description

Get Statistics:

- Method: `get_statistics()`

- Parameters: None
- Returns: Dictionary with total transactions, successful, failed, revenue, success rate

---

Architecture Overview

The vending machine system employs a clean layered architecture that systematically separates concerns across four distinct tiers, ensuring maintainability, testability, and adherence to the Single Responsibility Principle.

User Interface Layer (`ui/`)

At the highest level resides the User Interface Layer, implemented as a command-line interface through `Display`, `MenuHandler`, and `AdminHandler` classes. This layer handles all user interactions, menu displays, and input validation, serving as the system's presentation facade. It is designed to be easily replaceable with graphical or web interfaces without affecting core business logic.

Service Layer (`services/`)

Beneath the UI layer lies the Service Layer, which encapsulates cross-cutting concerns and external integrations. This layer includes:

- `PaymentService` abstraction with `CashPaymentService` implementation
- `TransactionService` responsible for recording, tracking, and reporting all transactions

The service layer acts as an intermediary that translates business operations into technical implementations while maintaining loose coupling between components.

Core Business Logic Layer (`core/`)

The foundation of the system is the Core Business Logic Layer, where the primary vending machine functionality resides. The `VendingMachine` class orchestrates all operations—money handling, item selection, inventory management, and purchase processing. This tier implements the essential business rules and workflows that define how a vending machine operates.

Domain Model Layer (`models/`)

Supporting all layers is the Domain Model Layer, which defines the fundamental entities and their relationships:

- Abstract `Drink` base class with `Soda`, `Juice`, `Water` implementations

- `Inventory` class for stock management

These domain objects encapsulate the core data structures and behaviors that represent the problem domain.

Class Relationships

Inheritance Hierarchy

The inheritance hierarchy centers on the abstract `Drink` base class, which defines the common interface and default implementations for all beverage types. This abstract class enforces polymorphism through its `get_description()` and `get_category()` abstract methods while providing concrete implementations for common properties.

Drink (Abstract)

├── Soda (carbonated beverages with is_diet attribute)

├── Juice (fruit-based drinks with fruit_type attribute)

└── Water (hydration products with is_sparkling attribute)

This hierarchical structure enables uniform treatment of diverse drink types while allowing specialized behaviors through method overriding and extension.

Composition Relationships

The central `VendingMachine` class maintains composition relationships with three key collaborators:

- **has-an** `Inventory` instance that manages all drink stock
- **has-a** `PaymentService` dependency that processes financial transactions
- **has-a** `TransactionService` component that records purchase history

The `Inventory` class itself contains multiple `Drink` objects organized by product codes (e.g., "A1", "B2"), creating a container-contained relationship that enables efficient item retrieval and stock tracking.

Sequence Flows

Successful Purchase Flow

1. **Insert Money**: Client calls `insert_money(2.00)` on VendingMachine
    - Machine updates internal balance tracker
    - Returns new total: `2.00`
2. **Select Item**: Client calls `select_item("A1")`
    - Machine delegates to Inventory via `get_item("A1")`
    - Inventory retrieves Drink object (Cola)
    - Machine checks stock via `has_stock("A1")` → True
    - Machine checks balance vs price: `2.00 >= 1.50` → True
3. **Process Payment**: Machine calls `deduct(1.50)` on PaymentService
    - PaymentService calculates change: `2.00 - 1.50 = 0.50`
    - Returns (True, 0.50)
4. **Update Inventory**: Machine calls `decrement_stock("A1")`
    - Inventory reduces quantity by 1
5. **Record Transaction**: Machine creates success transaction
    - Returns (True, "Dispensing Cola. Change: $0.50", transaction_dict)

Error Handling Flow (Insufficient Funds)

1. **Insert Money**: Client calls `insert_money(1.00)`
    - Balance updated to `1.00`
2. **Select Item**: Client calls `select_item("A1")` (price: $1.50)
    - Machine checks balance: `1.00 < 1.50` → Insufficient
3. **Handle Error**:
    - Machine records failed transaction
    - Returns (False, "Insufficient funds. Cola costs $1.50", None)

- Balance remains at `1.00` (no funds deducted)
- Inventory unchanged

This error flow demonstrates proper failure isolation: the payment failure doesn't corrupt inventory state, meaningful feedback is provided, and the system remains in a consistent, recoverable state.

Testing Summary

Test Suite	**Tests**	**Coverage**
test_drinks.py	15	Drink hierarchy, polymorphism
test_inventory.py	18	Stock management, queries
test_payment.py	12	Payment processing, refunds
test_transactions.py	14	Transaction recording, statistics
test_machine.py	18	Core controller logic
test_integration.py	10	End-to-end workflows
Total	**87**	**100% pass rate**

Conclusion

The Vending Machine OOP System demonstrates professional software engineering practices through its implementation of object-oriented principles, clean architecture, comprehensive testing, and robust error handling. The system provides a solid foundation that can be extended with additional features such as credit card payments, database persistence, web interfaces, or graphical user interfaces while maintaining the core architectural integrity.

The project successfully showcases:

- All four OOP pillars (Encapsulation, Abstraction, Inheritance, Polymorphism)
- SOLID principles throughout the architecture
- Clean separation of concerns across layers
- Comprehensive test coverage with 87 passing tests

- Professional documentation and code quality

Author: *Ayanda Phaketsi.*