Pharmacy Management System: OOP Project Report

Executive Summary

The Pharmacy Management System is a C++ application that demonstrates core object-oriented programming concepts through a practical inventory and billing solution. Using inheritance, polymorphism, and encapsulation, the system manages multiple medicine types, processes purchases, and generates bills. This report details the project architecture, implementation, and how fundamental OOP principles are applied in a real-world context .

1. Introduction

Managing a pharmacy requires tracking diverse medicine types, maintaining accurate stock levels, and processing customer transactions efficiently. Manual systems are error-prone and don't scale well. This project develops a digital Pharmacy Management System using object-oriented design in C++ to address these challenges.

The system allows staff to add different medicine categories to inventory, display available medicines, process purchases with stock validation, and generate itemized bills. By leveraging OOP principles, the solution is modular, maintainable, and easy to extend for future requirements. The architecture demonstrates how real business problems can be solved through well-structured code design.

2. Motivation

Business Perspective: Pharmacies need reliable digital systems to manage operations at scale. Manual inventory tracking leads to stockouts, overstocking, and lost sales. A computerized system ensures quick medicine lookups, real- time stock updates, and accurate billing—critical for customer satisfaction and business efficiency.

Educational Perspective: This project provides hands-on experience with core OOP concepts that define professional software development. Building a complete system from design to execution helps students understand how abstract programming concepts translate into practical solutions. The pharmacy domain makes these concepts concrete and relatable, far more effective than textbook examples alone.

3. Proposed Solution

3.1 System Architecture

The system uses four core classes organized in a clear hierarchy:

Medicine (Abstract Base Class):

```
class Medicine {
protected:
    string name;
    int quantity;
    double price;

public:
    Medicine(string n, int q, double p)
        : name(n), quantity(q), price(p) {}

    virtual void showDetails() = 0; // pure virtual function

    string getName() {
        return name;
    }
    int getQuantity() {
        return quantity;
    }
    double getPrice() {
        return price;
    }

    void updateQuantity(int q) {
        quantity = q;
    }

    bool reduceStock(int q) {
        if (q <= quantity) {
            quantity -= q;
            return true;
        }
        return false;
    }

    virtual ~Medicine() {}
};
```

The abstract base class defines a contract requiring all medicine types to implement showDetails(). It manages common data (name, quantity, price) and provides controlled access through methods like reduceStock(), which ensures inventory never goes negative. Protected members allow derived classes to access these attributes while preventing direct external modification.

Derived Classes (Antibiotic & Painkiller):

```
class Antibiotic : public Medicine {
public:
    Antibiotic(string n, int q, double p)
        : Medicine(n, q, p) {}

    void showDetails() override {
        cout << "----------------------------------------" << endl;
        cout << " Medicine Type : Antibiotic" << endl;
        cout << " Name          : " << name << endl;
        cout << " Stock         : " << quantity << endl;
        cout << " Price         : $" << fixed << setprecision(2) << price << endl;
        cout << "----------------------------------------" << endl;
    }
};

// Derived Class: Painkiller
class Painkiller : public Medicine {
public:
    Painkiller(string n, int q, double p)
        : Medicine(n, q, p) {}

    void showDetails() override {
        cout << "----------------------------------------" << endl;
        cout << " Medicine Type : Painkiller" << endl;
        cout << " Name          : " << name << endl;
        cout << " Stock         : " << quantity << endl;
        cout << " Price         : $" << fixed << setprecision(2) << price << endl;
        cout << "----------------------------------------" << endl;
    }
};
```

Both classes inherit from Medicine and override showDetails() to provide type-specific formatting. This demonstrates **inheritance** and **polymorphism**—when the system calls m->showDetails() through a base class pointer, the correct derived class method executes at runtime.

Bill Class:

```
// Billing Class
class Bill {
    double totalAmount = 0;

public:
    void addItem(double price, int qty) {
        totalAmount += (price * qty);
    }
    void generate() {
        cout << "\n---------------- BILL ----------------" << endl;
        cout << " Total Payable Amount : $" << fixed << setprecision(2) << totalAmount << endl;
        cout << "--------------------------------------" << endl;
        cout << " Thank you for your purchase!" << endl;
    }
};
```

This class encapsulates billing logic separately from medicine management, following the **Single Responsibility Principle**. It calculates costs and generates formatted receipts.

Pharmacy Class (Coordinator):

```
class Pharmacy {
    vector<Medicine*> items;

public:
    void addMedicine(Medicine* m) {
        items.push_back(m);
        cout << "Medicine added successfully." << endl;
    }

    void showAll() {
        if (items.empty()) {
            cout << "\nNo medicines in stock." << endl;
            return;
        }

        cout << "\n============ Medicine List ============" << endl;
        for (auto m : items) {
            m->showDetails();
        }
    }

    void purchaseMedicine() {
        string med;
        int qty;
        Bill bill;

        cout << "\nEnter medicine name to purchase: ";
        cin >> med;
        cout << "Enter quantity: ";
        cin >> qty;

        for (auto m : items) {
            if (m->getName() == med) {
                if (m->reduceStock(qty)) {
                    bill.addItem(m->getPrice(), qty);
                    cout << "\nPurchase successful!" << endl;
                } else {
                    cout << "\nInsufficient stock!" << endl;
                }
                bill.generate();
                return;
            }
        }
        cout << "\nMedicine not found in inventory." << endl;
    }
};
```

The Pharmacy class acts as the main coordinator, managing a collection of medicines using a vector of base class pointers. This design enables **polymorphic behavior at scale**—the container can hold any derived medicine type, and calling methods through base class pointers invokes the correct derived implementations automatically.

3.2 Key OOP Concepts Applied

Inheritance: Antibiotic and Painkiller inherit from Medicine, creating an is-a relationship where both are specialized medicine types. This enables code reuse and a clear classification hierarchy.

Polymorphism: Virtual functions allow the system to treat different medicine types uniformly. When showAll() iterates through medicines and calls showDetails(), the runtime type determines which implementation executes—no type checking or casting required.

Encapsulation: Data members are protected/private with public methods providing controlled access. Stock reduction must go through reduceStock(), which validates availability, preventing invalid states.

Abstraction: The Medicine class cannot be instantiated directly; users must create concrete Antibiotic or
Painkiller objects. This forces thinking about specific implementations from the start.

4. Implementation and Workflow

4.1 Main Function

```
int main() {
    Pharmacy ph;
    int choice;

    do {
        cout << "\n========= Pharmacy Management System =========" << endl;
        cout << "1. Add Medicine" << endl;
        cout << "2. Show All Medicines" << endl;
        cout << "3. Purchase Medicine" << endl;
        cout << "4. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        if (choice == 1) {
            int type, qty;
            double price;
            string name;

            cout << "\nSelect Type: 1. Antibiotic  2. Painkiller: ";
            cin >> type;
            cout << "Enter Name: ";
            cin >> name;
            cout << "Enter Quantity: ";
            cin >> qty;
            cout << "Enter Price: ";
            cin >> price;

            if (type == 1)
                ph.addMedicine(new Antibiotic(name, qty, price));
            else
                ph.addMedicine(new Painkiller(name, qty, price));
        }

        else if (choice == 2) {
            ph.showAll();
        }

        else if (choice == 3) {
            ph.purchaseMedicine();
        }

    } while (choice != 4);

    cout << "\nThank you for using the Pharmacy Management System!" << endl;
    return 0;
}
```

4.2 Operational Workflow

Adding Medicine: Users select medicine type and enter details. The system creates the appropriate derived class object using `new` and stores it as a base class pointer in the pharmacy's vector.

Displaying Inventory: The system iterates through all medicines and calls `showDetails()`. Through polymorphism, each medicine displays its type-specific information correctly.

Processing Purchase: The system searches for the requested medicine, validates stock availability, reduces inventory atomically, calculates cost, and generates a bill. Error messages handle invalid scenarios (medicine not found, insufficient stock).

4.3 Sample Output

```
========= Pharmacy Management System =========
1. Add Medicine
2. Show All Medicines
3. Purchase Medicine
4. Exit
Enter your choice: 1

Select Type: 1. Antibiotic  2. Painkiller: 1
Enter Name: Amoxicillin
Enter Quantity: 20
Enter Price: 30.50
Medicine added successfully.

========= Pharmacy Management System =========
1. Add Medicine
2. Show All Medicines
3. Purchase Medicine
4. Exit
Enter your choice: 2

=========== Medicine List ===========
-----------------------------------------
 Medicine Type : Antibiotic
 Name          : Amoxicillin
 Stock         : 20
 Price         : $30.50
-----------------------------------------

========= Pharmacy Management System =========
1. Add Medicine
2. Show All Medicines
3. Purchase Medicine
4. Exit
Enter your choice: 3

Enter medicine name to purchase: Amoxicillin
Enter quantity: 5

Purchase successful!

---------------- BILL ----------------
 Total Payable Amount : $152.50
--------------------------------------
 Thank you for your purchase!
```

5. Results and Key Features

This project uses **dynamic inventory management**, where medicines can be added while the program is running. The use of a vector helps the system store any number of medicines, from a few to thousands, without changing the code.

It also has **type flexibility**, which means right now it works for Antibiotics and Painkillers, but new types like Vitamins can be added easily by making a new class from the Medicine class. The system also has **strong transaction checking**, which makes sure that before buying, the medicine exists, there is enough stock, and the quantity is reduced only when everything is correct.

This keeps the data right and avoids mistakes. Lastly, **good memory management** is used with virtual destructors so that all medicines are properly deleted and no memory is wasted when the program runs for a long time.

6. Conclusion

This project successfully demonstrates core OOP principles in solving a real-world business problem. Through the use of inheritance, polymorphism, encapsulation, and abstraction, we created a flexible system that manages pharmacy operations effectively while remaining maintainable and extensible.

The modular design—separating medicine management, billing, and user interaction—reflects professional software architecture practices. The ability to handle different medicine types uniformly through base class interfaces while allowing type-specific behaviors showcases polymorphism's power and elegance.

Future enhancements could include persistent storage (files/databases), advanced search filters, medicine expiry tracking, graphical user interfaces, or multi-user networking support. The current architecture provides a solid foundation where such extensions require minimal changes to core logic, validating the effectiveness of OOP design principles.

7. References

1) E. Balagurusamy – Object-Oriented Programming with C++
2) GeeksforGeeks – OOP Design Concepts
3) Oracle Docs – Polymorphism and Abstraction
4) TutorialsPoint – UML Class Diagrams
5) cppreference.com – Standard C++ Library Reference