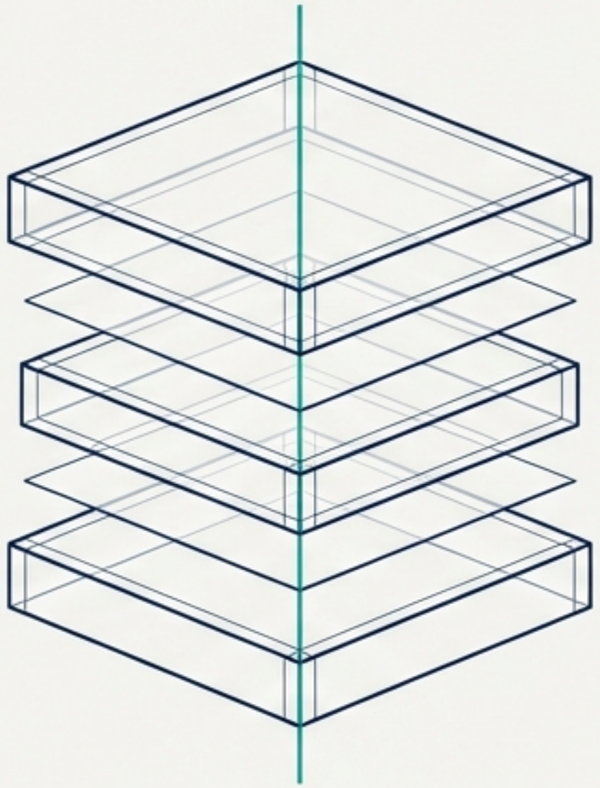
C++ Inventory & Sales System: A Study in Layered Architecture



Deconstructing a command-line application to reveal the principles of robust, maintainable, and testable software design.

The Mission: Build a Simple, Robust, and Persistent Inventory System

Manage Inventory

Track items with core attributes like name, quantity, and price.

Process Sales

Record transactions, update stock levels, and calculate profit.

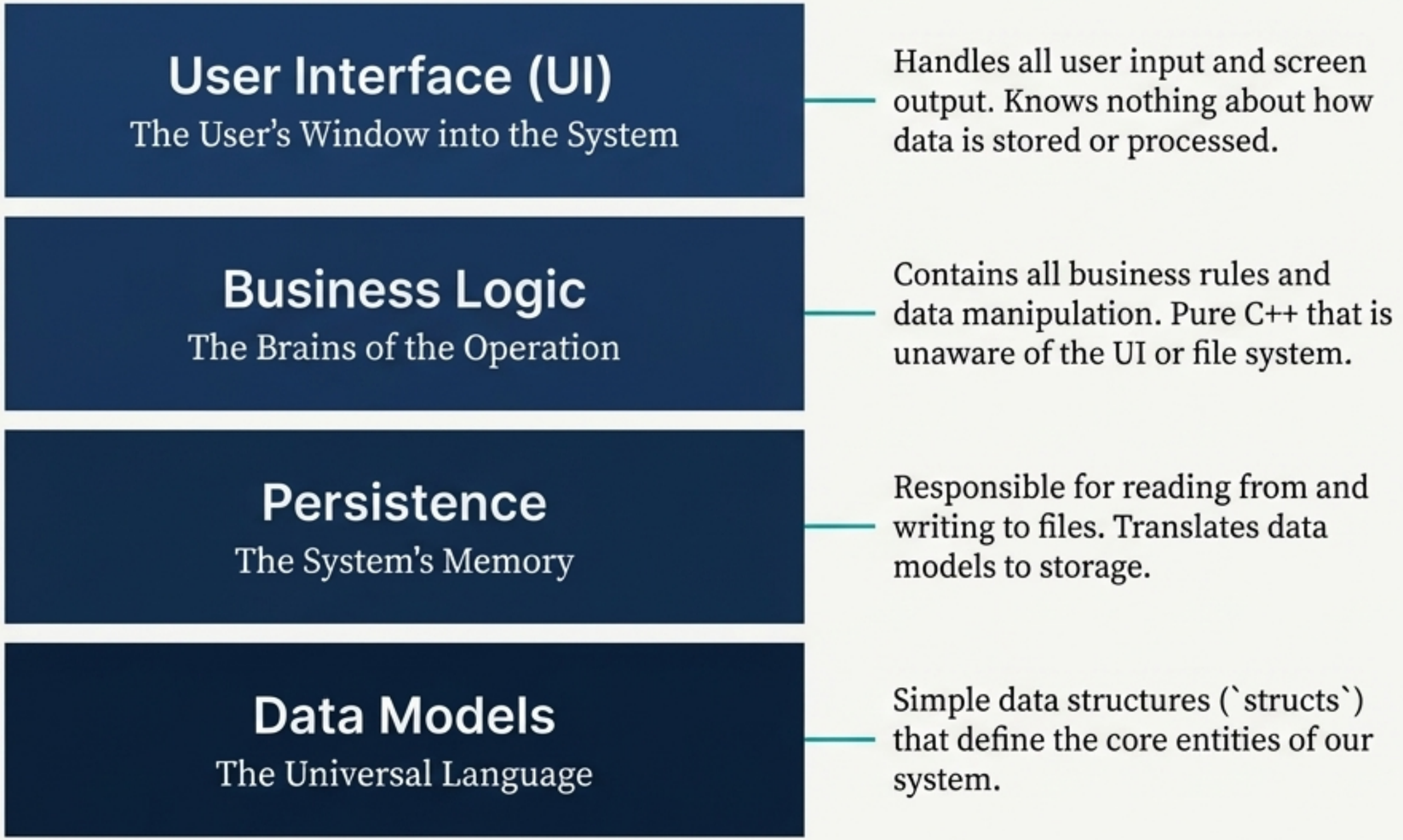
Ensure Persistence

Data must survive application restarts.

Operate via Console

A straightforward command-line interface for all user interactions.

Our Architectural Blueprint is Built on Separation of Concerns



Layer 1: The Data Foundation Defines the System's Core Entities

At the very core, our system **needs a common language to describe its two main concepts: inventory items** and **sales records.** We use simple C++ `structs` for this. They are lightweight, transparent, and serve as the **single source** of truth for what our data looks like. These structs are passed between all layers, acting as the lifeblood of the application.

`Item`

`id`
`name`
`quantity`
`purchase_price`
`selling_price`

Represents a product in the inventory. Contains its ID, name, quantity, and pricing.

`Sale`

`id`
`item_id`
`quantity_sold`
`profit`
`date_sold`

Represents a single transaction. A snapshot containing what was sold, how many, the profit, and when it occurred.

Code: The Data Models are Clean and Self-Documenting

```
/**
 * @brief Represents an item in the inventory.
 */
struct Item {
    int id;                     ///< Unique ID of the item
    string name;                ///< Name of the item
    string size_color;          ///< Size or Color variant
    int quantity;               ///< Current stock quantity
    double purchase_price;      ///< Cost price
    double selling_price;       ///< Selling price
};
/**
 * @brief Represents a sales record.
 */
struct Sale {
    int id;                     ///< Unique ID of the sale
    int item_id;                ///< ID of the item sold
    string item_name;           ///< Name of the item sold (snapshot)
    int quantity_sold;          ///< Quantity sold
    double profit;              ///< Profit made from this sale
    string date_sold;           ///< Timestamp of the sale
};
```

No complex classes or methods. These are pure data containers, making them easy to understand, serialize, and use across the application.

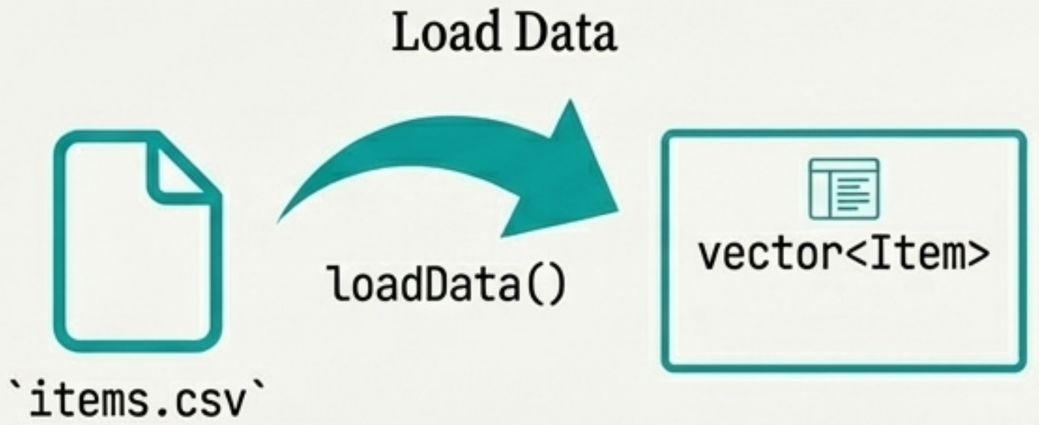
Layer 2: The Persistence Engine Uses Simple, Human-Readable Files

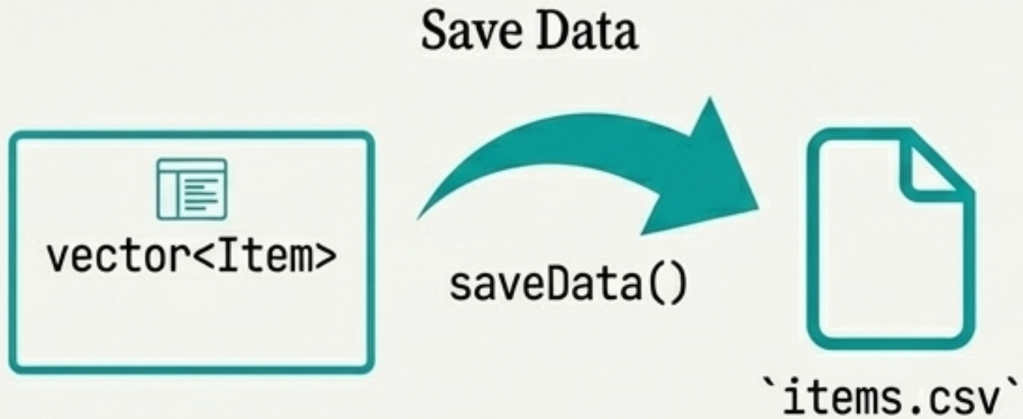
To ensure data survives between sessions, we need a persistence mechanism. Instead of a heavy database, we opted for a simple and robust solution: CSV files.

This approach keeps the system lightweight and portable. The data files (`items.csv`, `sales.csv`) can be inspected or even edited with a simple text editor.

Key Functions

- `loadData()`: Reads from the CSV files on startup, populating the in-memory `vector<Item>` and `vector<Sale>`.
- `saveData()`: Writes the current state of the in-memory vectors back to the CSV files before exiting.





Code: Persistence is a Direct Translation of In-Memory Data

```
// In saveData()
ofstream itemFile(ITEMS_FILE);
if (itemFile.is_open()) {
    for (const auto& item : items) {
        itemFile << item.id << ","
                 << item.name << ","
                 << item.size_color << ","
                 << item.quantity << ","
                 << item.purchase_price << ","
                 << item.selling_price << "\n";
    }
    itemFile.close();
}
```

Iterates through the live, in-memory data.

Each `Item` struct is deconstructed into a single comma-separated line.

Layer 3: The Logic Core is Decoupled and Testable

This is the heart of the application. The logic layer contains all the business rules, calculations, and state manipulations. Crucially, these functions are 'pure.' They operate only on the data models passed to them. They don't know about `cout`, `cin`, or file streams. This separation is the key to maintainability. We can change the UI or the database without ever touching this core logic.

Adding and deleting items (`logic_addItem`, `logic_deleteItem`)

Handles the logic for creating new inventory records and removing existing ones based on user input.

Updating stock (`logic_updateItem`)

Manages inventory quantities, ensuring accurate stock levels are maintained after transactions.

Processing sales and calculating profit (`logic_sellItem`)

Executes sales transactions, updates stock, and computes profit margins for business analysis.

Because this layer is self-contained, every function can be independently unit-tested.

Code: `logic_sellItem` Encapsulates Critical Business Rules

```
int logic_sellItem(int id, int qty, double& profitOut) {
    auto it = find_if(items.begin(), items.end(), ...);

    if (it == items.end()) return 1; // Not found

    if (qty > it->quantity) return 2; // Not enough stock

    double profit = (it->selling_price - it->purchase_price) * qty;
    it->quantity -= qty;

    // Record sale
    sales.push_back({nextSaleId++, it->id, it->name, ...});

    profitOut = profit;
    return 0; // Success
}
```

1 Validate Existence

First, ensure the item exists.

2 Validate Inventory

Next, check for sufficient stock before proceeding.

3 Update State

The core operation: decrement the item's quantity in memory.

4 Create Record

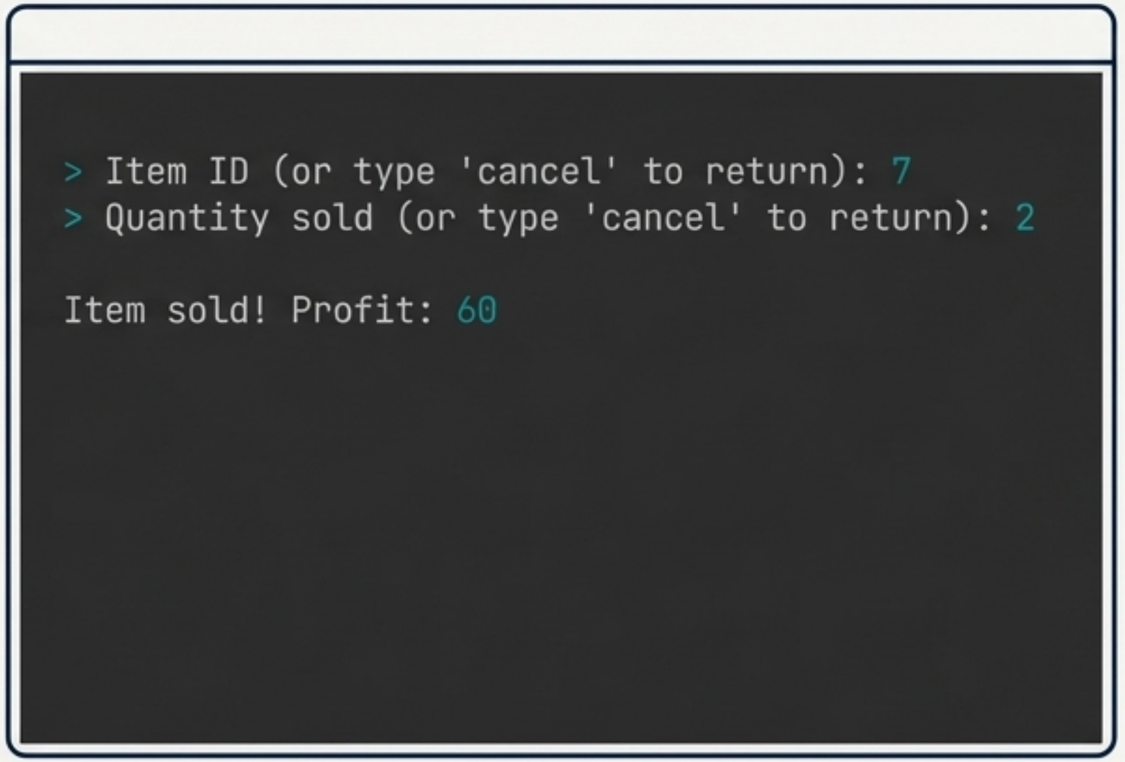
A new `Sale` record is created and added to the sales log.

Layer 4: The User Interface is a Thin Layer for Input and Output

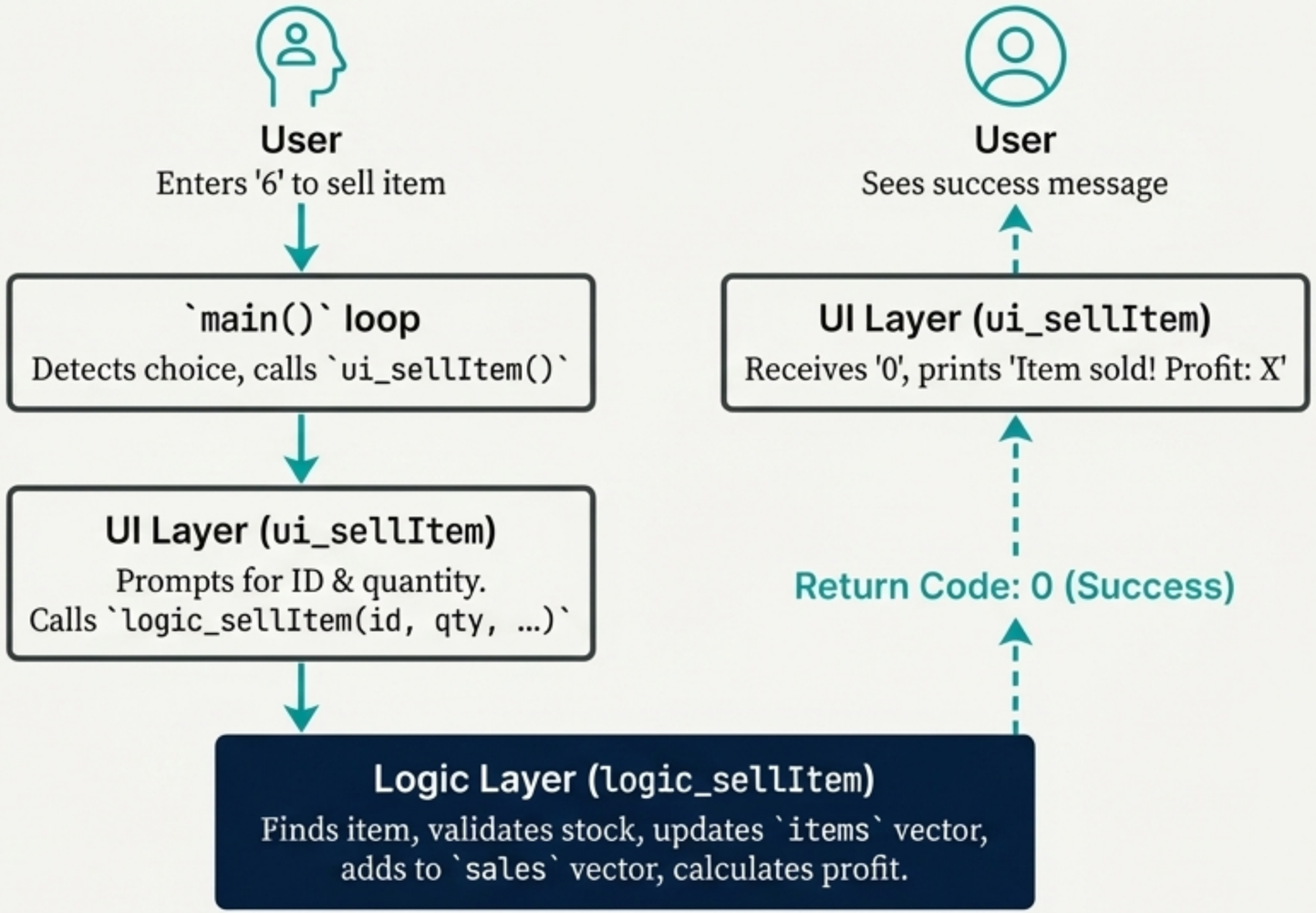
The UI layer's only job is to communicate with the user. It manages the command-line menu, prompts for input, and displays results.

1. Gather data from the user.
2. Call the appropriate logic function with that data.
3. Interpret the result and display a user-friendly message.

Naming Convention: A clear `ui_*` prefix (e.g., `ui_addItem`, `ui_sellItem`) distinguishes these functions from the `logic_*` core.



Tying It All Together: The Journey of a Sale



This clean handoff between layers ensures that user interface concerns and business rules never mix.

The Support System: Utilities and Helpers Ensure Robustness

Good software isn't just about the major architectural layers. A collection of small, focused helper functions keeps the main codebase **clean, readable, and resilient** to bad user input.

Input Parsing

`toInt()`
`toDouble()`

Safely convert user input strings to numbers without crashing.

```
try {
  // ...
} catch (...) {
  // ...
}
```

String Manipulation

`trim()`
`toLowerStr()`

Sanitize user input by removing whitespace and normalizing case.

```
s.find_first_not_of(" \t");
```

Convenience

`isCancel()`

Provides a consistent way for users to exit an operation.

```
return (t == "cancel");
```

The Conductor: The `main()` Loop Orchestrates the Application

The `main()` function acts as the central hub. Its primary role is to run a loop that displays the menu, captures the user's choice, and delegates the task to the correct `ui_*` function.

```
do {
    // ... (cout menu options) ...
    cout << "Choice: ";
    cin >> choice;

    switch (choice) {
        case 1: ui_addItem(); break;
        case 6: ui_sellItem(); break;
        case 10: saveData(); break;
        // ... etc ...
    }
} while (choice != 10);
```

Notice that `main` only calls `ui_*` functions or `saveData`. It has no direct access to the business logic, honoring the layered design.

Key Architectural Takeaways

Separation of Concerns

Each layer has a single, well-defined responsibility. The UI handles presentation, the Logic handles business rules, and Persistence handles storage. This makes the system easier to reason about and modify.

High Testability

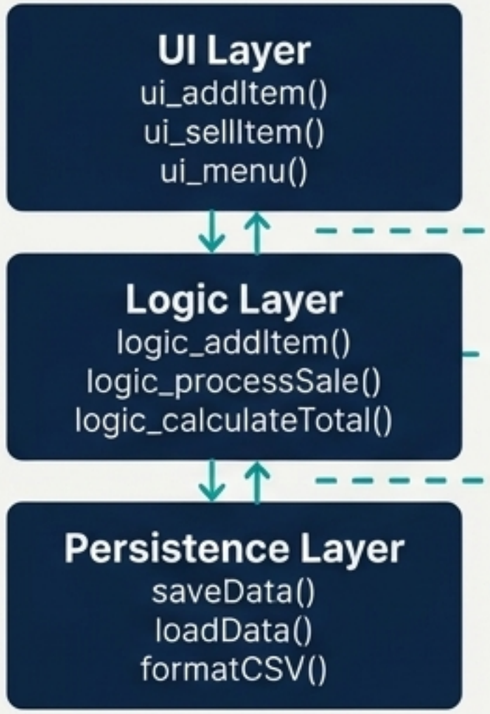
The `logic_*` functions are completely isolated from the outside world. They can be put under a suite of unit tests to verify their correctness without ever needing to run the main application or interact with the console.

Flexible Persistence

The choice of CSV is simple and effective. Because it's isolated in its own layer, the entire persistence mechanism could be swapped out for a SQL database or an API call with minimal changes to the rest of the application.

A Solid Foundation for Future Evolution

This layered architecture is more than just a way to organize code for a simple application; it's a professional pattern that enables future growth and complexity.



Swap the UI: The command-line interface could be replaced with a graphical UI (e.g., Qt) or a web-based API without changing a single line in the `logic_*` core.

Add Features: New business logic, like generating advanced reports, can be added to the logic layer without disrupting existing functionality.

Upgrade Persistence: The CSV file handlers could be swapped out to connect to a professional SQL database like PostgreSQL for better performance and scalability.

Good architecture isn't about solving today's problem; it's about making it easier to solve tomorrow's.