
**TOPIC :** Client contact CRUD Manager

**NAME :** Disha Krishnappa Moolya

**REGISTRATION No. :** 25BCY10003

# Introduction

This project implements a fundamental **Client Contact Management System** using Python. Its primary purpose is to provide a simple, reliable, and persistent way to handle core client data operations.

---

**Purpose and Functionality**

The project solves the problem of unstructured data management by offering a programmatic, standardized approach to handling contacts. The core functionality centers around **CRUD** (Create, Read, Update, Delete) operations:

- **Data Structure:** Contacts are represented by the **Contact class**, ensuring every record consistently stores an ID, name, phone, email, and company.
- **Management Logic:** The **ContactManager class** acts as the central engine, responsible for executing all CRUD commands and managing the collection of contacts.

**Technology and Persistence**

The system leverages the **Python Standard Library** for simplicity and effectiveness:

- **Language:** Python
- **Persistence:** It uses the **CSV file format** (Contacts_data.csv) for persistent storage. The built-in **csv module** handles all file input/output, ensuring data is automatically loaded upon starting and saved after any modification.

**Scope**

This is a **high-level, backend-only** project. It serves as an excellent foundational example of **Object-Oriented Programming (OOP)** and **file I/O**, designed for small-scale personal or demonstration use. It deliberately excludes complex features like a graphical user interface (GUI), searching, or advanced data validation, focusing solely on the core persistent CRUD logic.

# Problem Statement: Client Contact CRUD Manager

The core problem this project addresses is the **inefficient and disorganized management of client contact information**, specifically the lack of a simple, persistent, and standardized system for basic data operations.

---

**The Issue**

Organizations or individuals often manage vital contact data (names, phone numbers, emails, companies) using unstructured or manual methods like basic text files or simple, manually edited spreadsheets. This approach presents several key challenges:

1. **Lack of Structure and Error:** Data lacks a defined schema, making it difficult to guarantee consistency. It is highly prone to manual entry errors, duplication, and inconsistent formatting.
2. **Difficult Management:** There is no easy, programmatic way to perform essential actions such as:
    - Ensuring every new contact gets a unique ID.
    - Quickly retrieving a specific record.
    - Reliably updating or deleting outdated entries without risking data corruption.
3. **No Automated Persistence:** Changes rely entirely on manual saving, which increases the risk of losing recent modifications if the system isn't robustly designed to save upon every modification.

---

**The Solution (Project Goal)**

The goal of the Client Contact CRUD Manager is to develop a **simple, reliable, and persistent contact management system** that addresses the above inefficiencies.

The solution provides:

- A **structured data model** (Contact class) for guaranteed data consistency.
- A **centralized logic layer** (ContactManager class) to automate all management tasks (CRUD).
- **Automated Persistence** using a **CSV file** to reliably save and load data across program sessions.

# Functional Requirements for the Client Contact CRUD Manager

The functional requirements define what the system **must do** to meet the project's goal of managing contacts. They are based directly on the CRUD operations and data persistence needs.

---

- **Create Contact:** Allow input of **name, phone, email, and company** to create a new record.
- **Auto-ID Generation:** Automatically assign a **unique, sequential integer ID** to every new contact.
- **Retrieve All:** Provide the ability to retrieve and display **all** existing contact records.
- **Retrieve by ID:** Allow retrieval of a **single contact** using its unique ID.
- **Update Contact:** Allow **modification** of any contact attribute (name, phone, email, company) based on its ID.
- **Delete Contact:** Allow **permanent removal** of a contact record based on its ID.
- **Load Data: Load all contact data** from the Contacts_data.csv file upon startup.
- **Save Data: Automatically save all contacts** back to Contacts_data.csv after every successful Add, Update, or Delete operation.

# Non-functional Requirements

The system must meet the following quality attributes and constraints:

---

- **Reliability:** The system must ensure data integrity by maintaining unique contact IDs and handling file loading/saving gracefully, even if the CSV file is initially missing.
- **Performance:** All CRUD operations must execute quickly (negligible time) on small datasets.
- **Usability:** The code must be **clear, well-commented, and modular** to ensure ease of maintenance and readability.
- **Portability:** The system must rely **only on the Python Standard Library** (e.g., csv, os) to run without external dependencies on any major operating system supporting Python 3.
- **Security:** The scope is limited to **local, single-user operation**, and no complex security mechanisms (like authentication) are required.

# System Architecture

The Client Contact CRUD Manager employs a **simple, two-tier architecture** typical of small, file-based applications. This structure clearly separates the application logic from the data storage.

---

### 1. Application Layer (Logic Tier)

This layer contains all the code responsible for processing data, controlling application flow, and executing business logic. It operates in memory while the application is running.

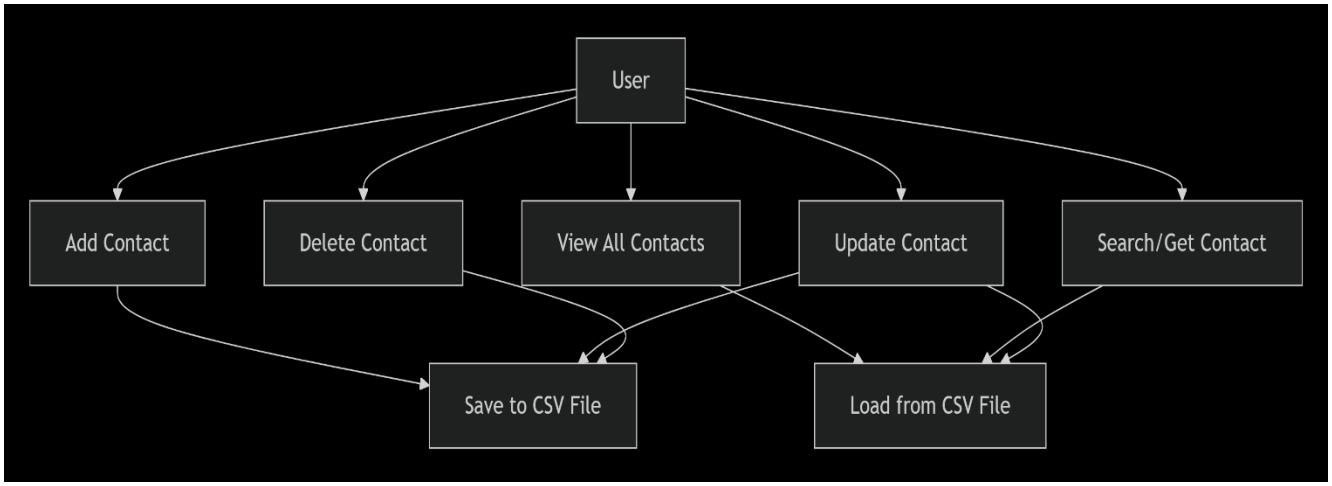
- **Data Model (Entity):** The **Contact Class** defines the standard structure for all contact records (ID, name, phone, etc.).
- **Business Logic/Controller:** The **ContactManager Class** acts as the central engine. It performs all the **CRUD** functions (Add, Get, Update, Delete) and manages the in-memory list of Contact objects.

### 2. Persistence Layer (Data Tier)

This layer is responsible for the permanent storage of data on the disk.

- **Storage Medium:** The **Contacts_data.csv** file serves as the flat-file database.
- **Data Access:** The **ContactManager** directly interacts with this file using the Python **csv module** to save and load data, ensuring that changes are persistent across different runs of the program.

# Use Case Diagram




**Use Case Descriptions:**

**Primary Actors:**

- **User** (the person interacting with the system)

**Core Use Cases:**

1. **Add Contact**
    - **Description**: User adds a new contact with name, phone, email, and company
    - **Pre-condition**: Contact manager is initialized
    - **Post-condition**: New contact is saved to CSV file
2. **View All Contacts**
    - **Description**: User retrieves and views all contacts in the system
    - **Pre-condition**: Contact file exists or contacts are loaded
    - **Post-condition**: Contacts are displayed to user
3. **Search/Get Contact**
    - **Description**: User searches for a specific contact by ID
    - **Pre-condition**: Contacts are loaded into memory
    - **Post-condition**: Contact details are returned (or None if not found)
4. **Update Contact**
    - **Description**: User modifies existing contact information
    - **Pre-condition**: Contact exists in the system
    - **Post-condition**: Updated contact is saved to CSV file
5. **Delete Contact**

- **Description**: User removes a contact from the system
- **Pre-condition**: Contact exists in the system
- **Post-condition**: Contact is removed and changes saved to CSV file

**System Use Cases (Automated):**

6. **Load from CSV File**
    - **Description**: System automatically loads contacts from CSV file on startup
    - **Trigger**: ContactManager initialization
7. **Save to CSV File**
    - **Description**: System automatically persists contact changes to CSV file
    - **Trigger**: After add, update, or delete operations

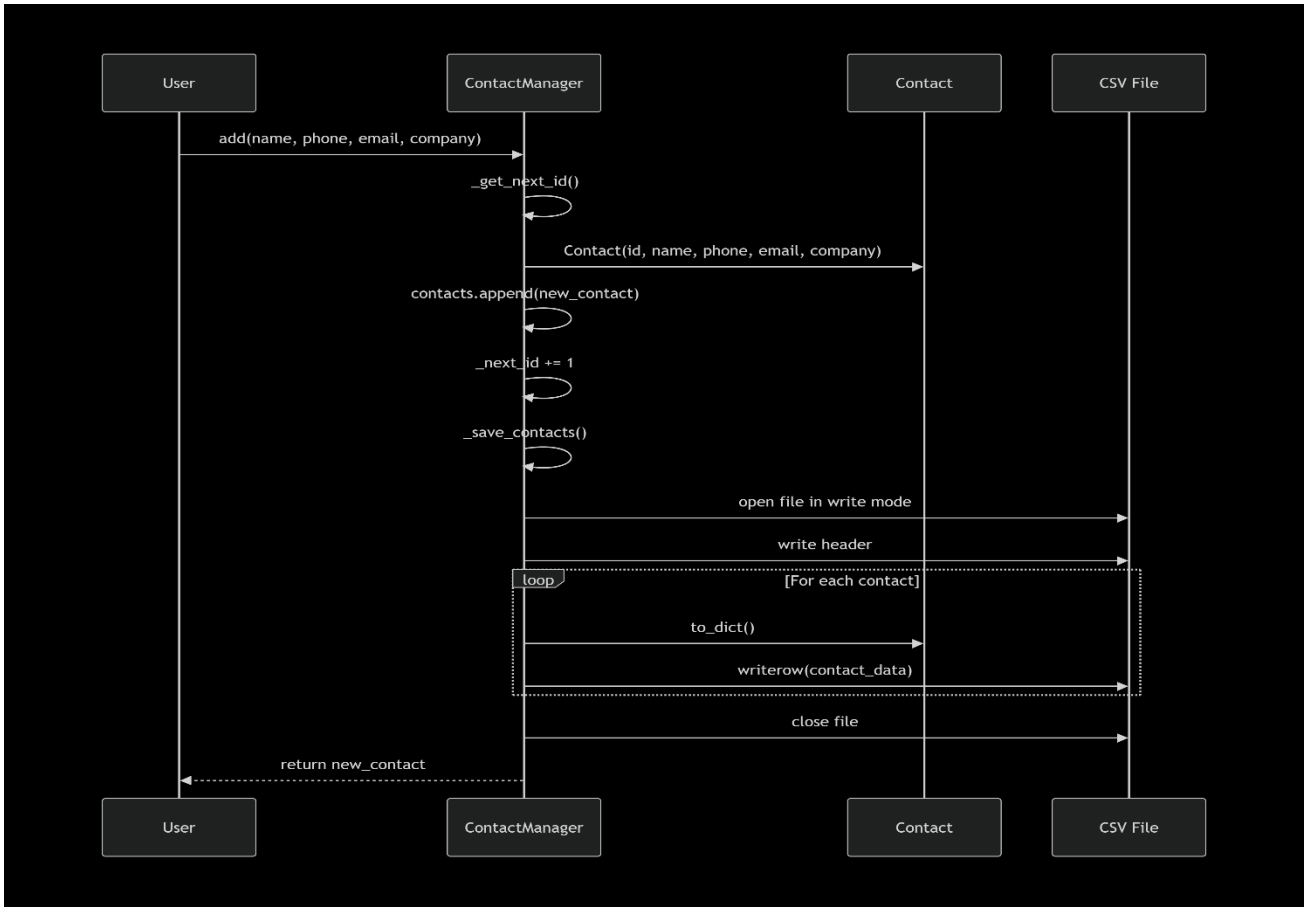
**Relationships:**

- **Include**: Update Contact **includes** Load from CSV File
- **Include**: Add/Update/Delete **include** Save to CSV File
- **Extend**: All view operations **extend** Load from CSV File

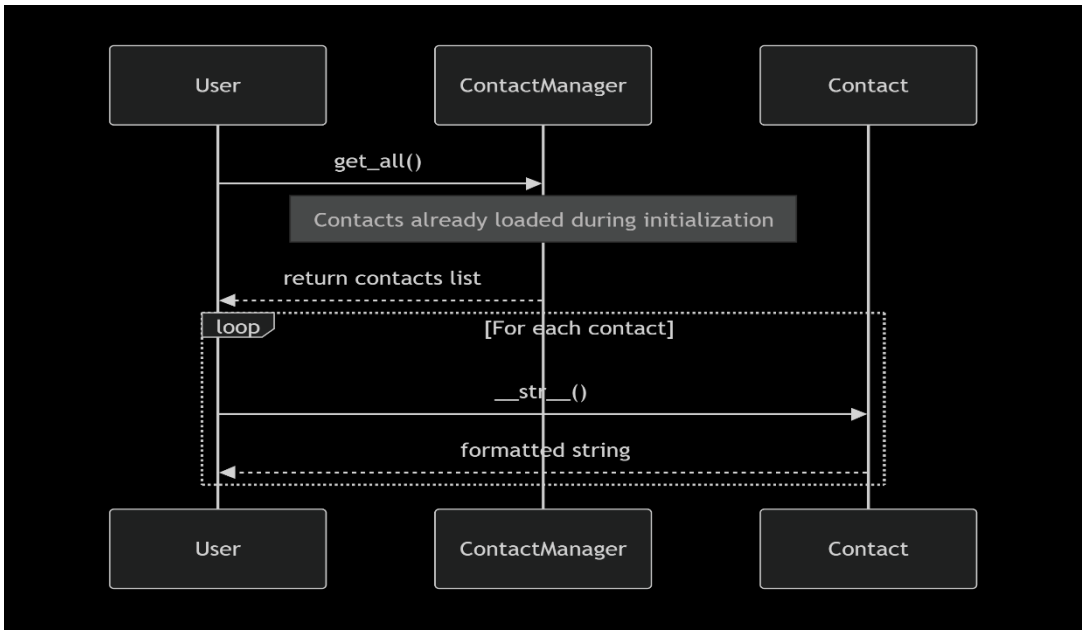
This diagram shows how users interact with the contact management system and how the system handles data persistence automatically.

# Sequence Diagram

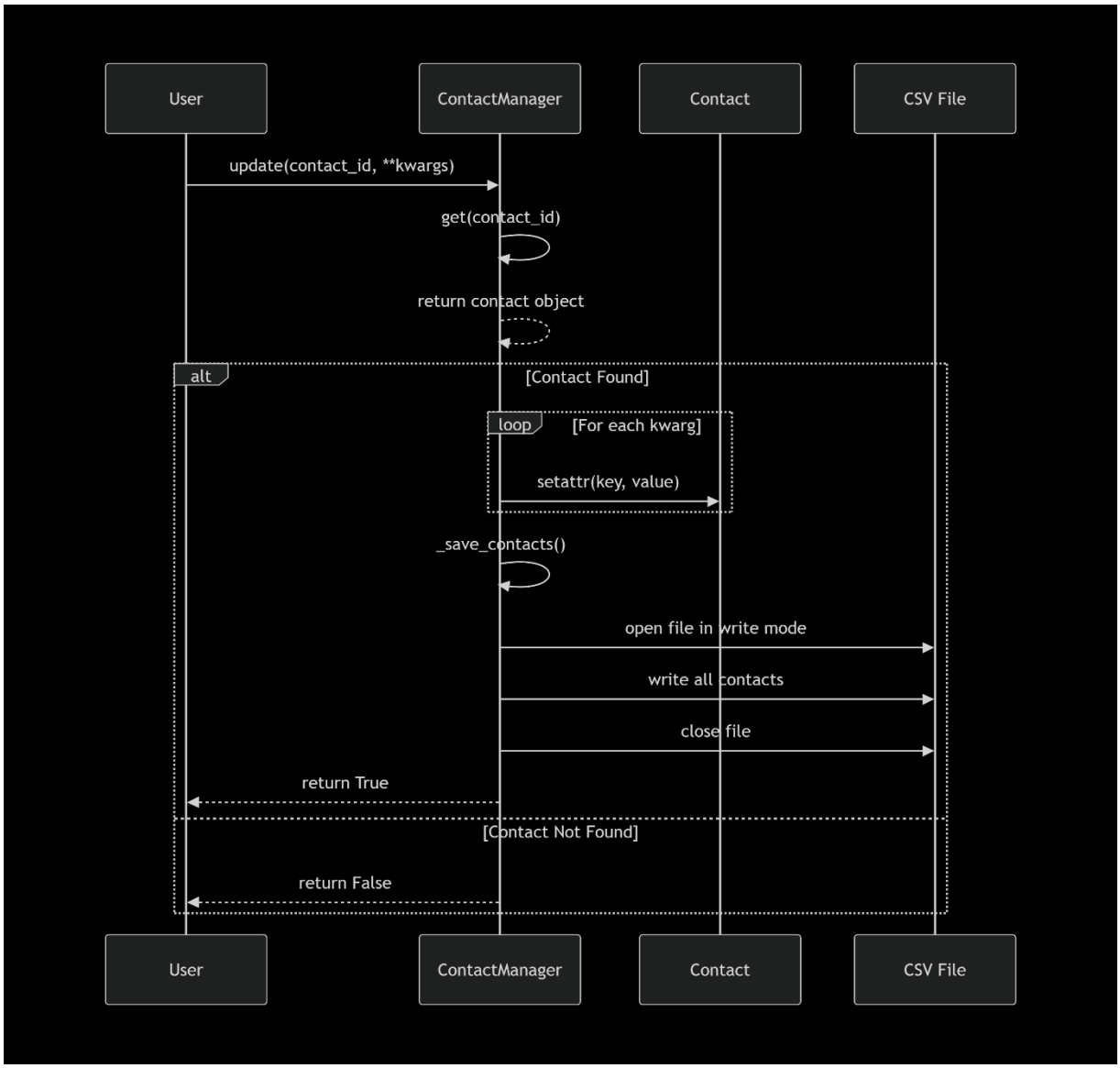
## 1. Add Contact Sequence Diagram



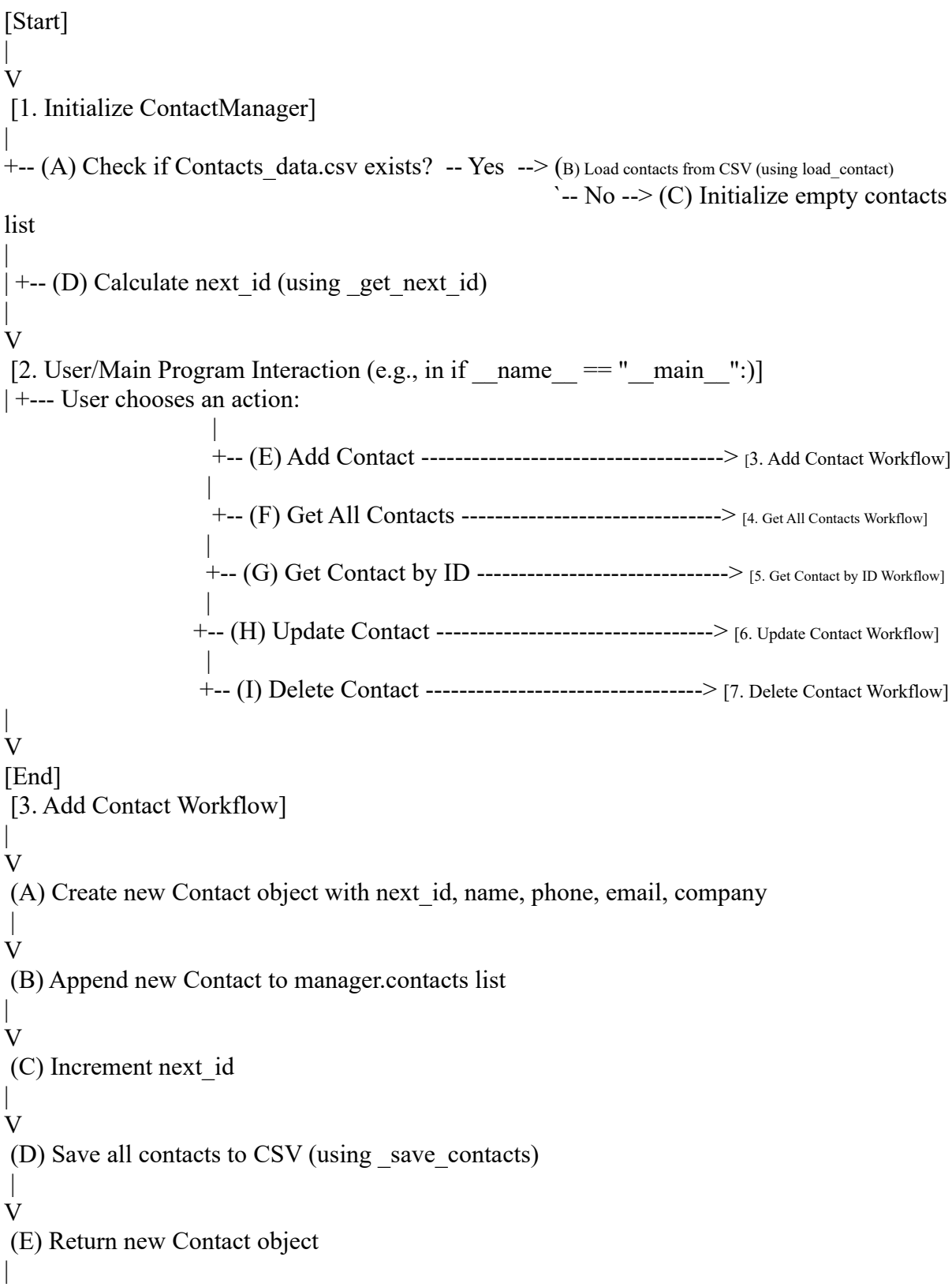

## 2. View All Contacts Sequence Diagram



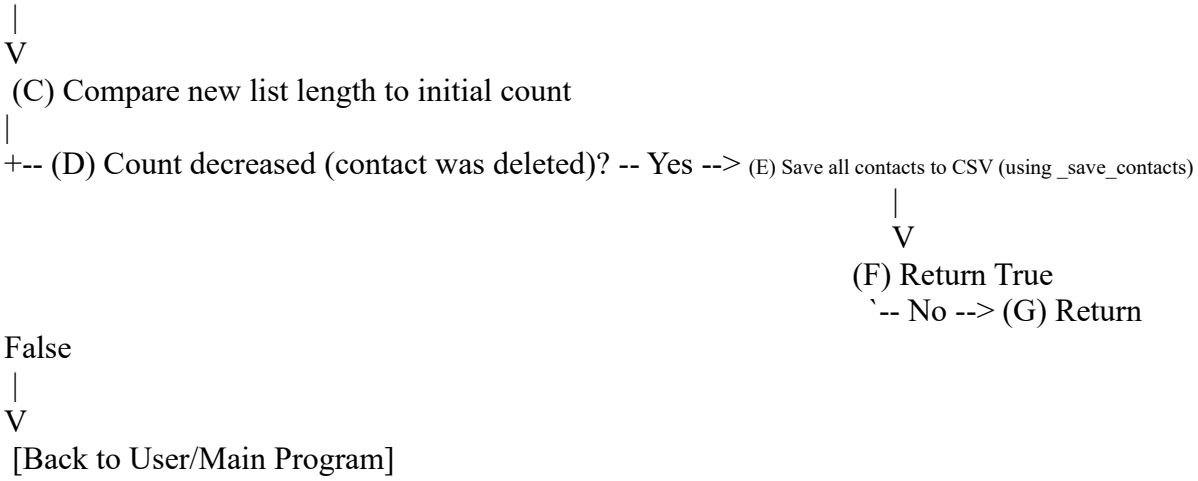
# 3. Update Contact Sequence Diagram



# Workflow Diagram: Client Contact CRUD Manage

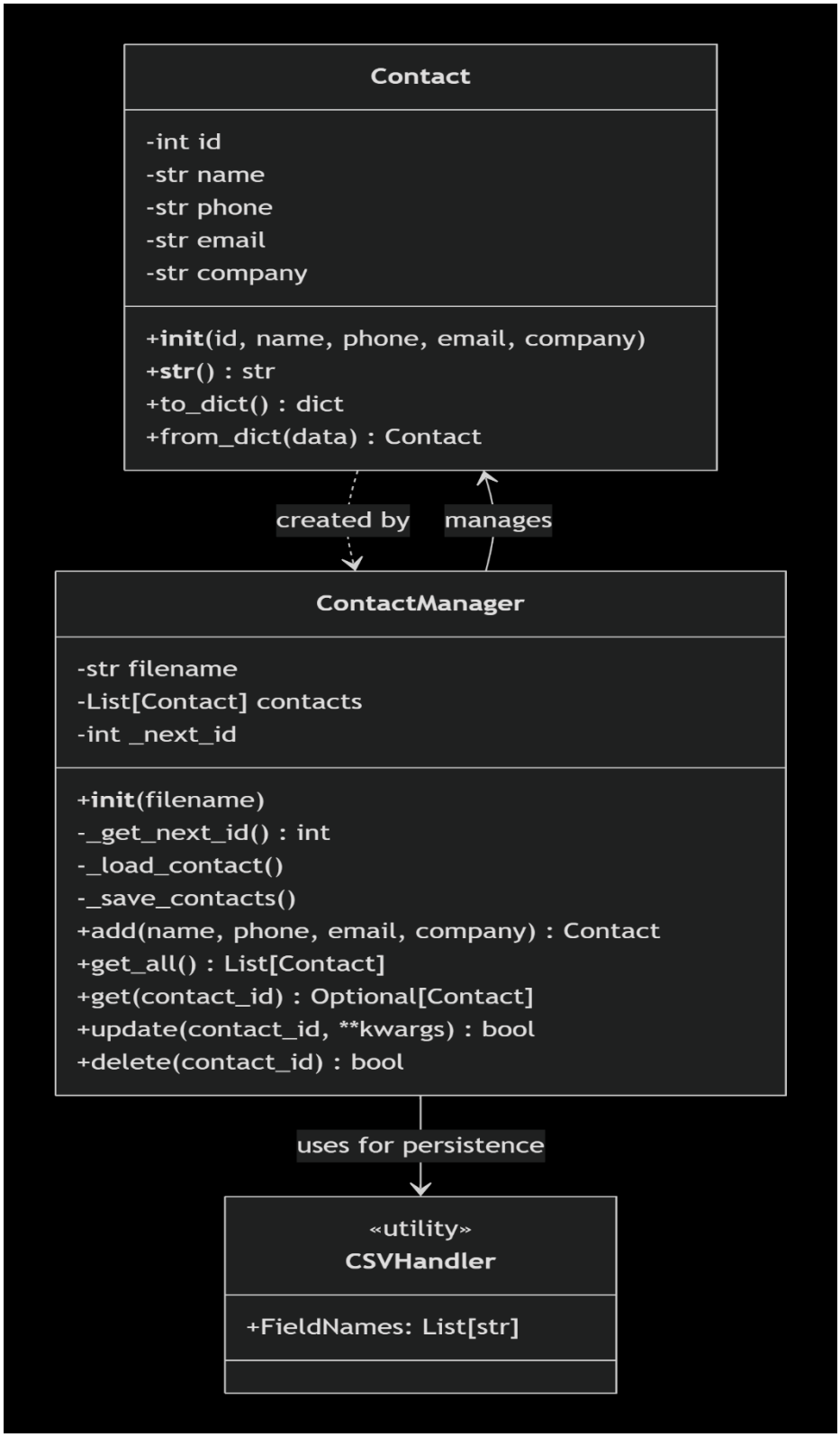


V
[Back to User/Main Program]
---
[4. Get All Contacts Workflow]
|
V
(A) Return manager.contacts list
|
V
[Back to User/Main Program]
---
[5. Get Contact by ID Workflow]
|
V
(A) Iterate through manager.contacts list
|
+-- (B) Find Contact with matching ID? -- Yes --> (C) Return found Contact object
`-- No --> (D) Return None
|
|
V[
Back to User/Main Program]
---
[6. Update Contact Workflow]
|
V
(A) Find Contact by ID (similar to "Get Contact by ID" workflow)
|
+-- (B) Contact found? -- No --> (C) Return False
`-- Yes --> (D) Iterate kwargs (key, value)
|
V
(E) For each kwarg, update corresponding Contact attribute usingsetattr()
|
V
(F) Save all contacts to CSV (using_save_contacts)
|
V
(G) Return True
|
V
[Back to User/Main Program]
[7. Delete Contact Workflow]
|
V
(A) Store initial count of contacts
|
V
(B) Filter manager.contacts, keeping only contacts whose ID does NOT match the given contact_id

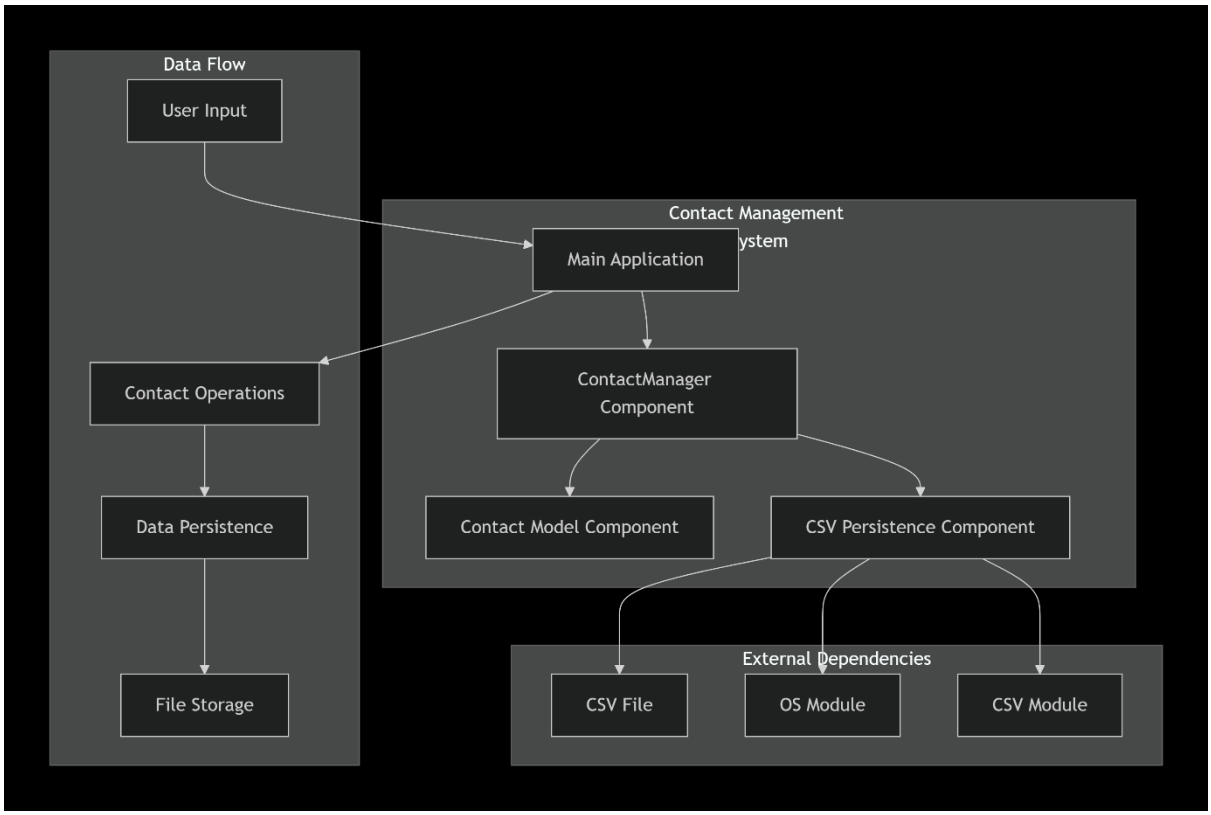

(C) Compare new list length to initial count
(D) Count decreased (contact was deleted)?
Yes
(E) Save all contacts to CSV (using _save_contacts)
(F) Return True
No
(G) Return False
[Back to User/Main Program]

# Class/Component Diagram

## 1. Class Diagram



# 2. Component Diagram



# ER Diagram


CONTACTS
int
id
PK
Primary Key
varchar
name
Contact Name
varchar
phone
Phone Number
varchar
email
Email Address
varchar
company
Company Name

# Design Decisions & Rationale

**1. Data Persistence**

- **Decision**: Use CSV file over database
- **Rationale**: Simple storage for small-scale contact management, no external dependencies, easy debugging

**2. ID Management**

- **Decision**: Auto-incrementing integer IDs managed in code
- **Rationale**: Simple unique identification, no database sequence required

**3. Class Structure**

- **Decision**: Separate Contact and ContactManager classes
- **Rationale**:
    - **Separation of Concerns**: Contact handles data, Manager handles operations
    - **Single Responsibility**: Each class has one clear purpose

**4. Data Loading Strategy**

- **Decision**: Load all contacts into memory on startup
- **Rationale**: Fast read operations, suitable for small datasets

**5. Update Mechanism**

- **Decision**: Generic update with **kwargs
- **Rationale**: Flexible partial updates without separate methods for each field

**6. Error Handling**

- **Decision**: Skip malformed rows during load
- **Rationale**: System remains usable even with some corrupt data

**7. File Operations**

- **Decision**: Rewrite entire file on every change
- **Rationale**: Simpler implementation, ensures data consistency

**8. Static Method**

- **Decision**: from_dict() as static method
- **Rationale**: Clean object creation without requiring Contact instance

# Implementation Details

The project is implemented using Python and adheres to **Object-Oriented Programming (OOP)** principles to separate the data model from the management logic.

---

**1. Data Model (Contact Class)**

- **Structure:** Defines the blueprint for a single record with attributes: id (int), name (str), phone (str), email (str), and company (str).
- **Conversion:** Includes methods (to_dict and from_dict) to easily convert between the Python object and the dictionary format required for file I/O.

**2. Management Logic (ContactManager Class)**

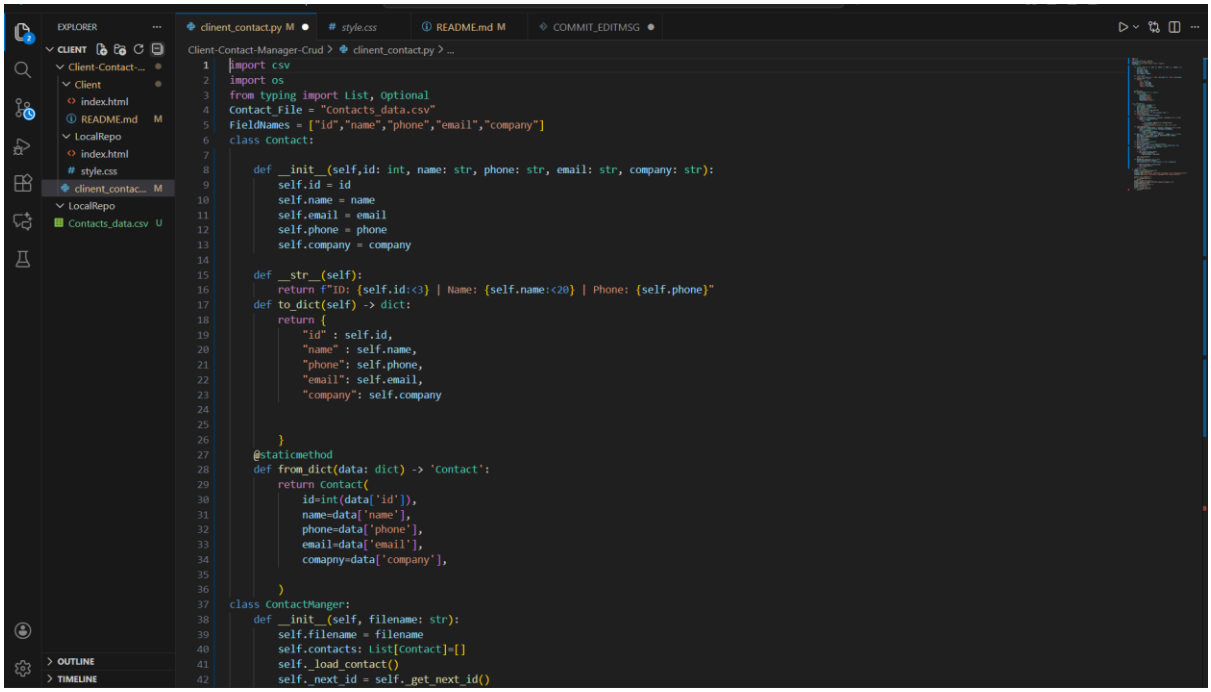
- **In-Memory Storage:** Uses a **Python list (self.contacts)** to hold all Contact objects currently loaded.
- **Persistence Handlers:**
    - _load_contact(): Uses **csv.DictReader** to read the Contacts_data.csv file and populate self.contacts.
    - _save_contacts(): Uses **csv.DictWriter** to write the entire self.contacts list back to the CSV file, ensuring persistence after every modification.
- **ID Management:** The _get_next_id() method automatically calculates the next available unique ID based on the maximum existing ID.
- **CRUD Implementation:**
    - **Add:** Creates a new Contact using the next ID and calls _save_contacts.
    - **Update:** Uses self.get(id) to find the contact and **setattr()** to dynamically update attributes based on keyword arguments (**kwargs).
    - **Delete:** Uses a **list comprehension** to filter out the contact with the matching ID, then calls _save_contacts.

**3. File System Interaction**

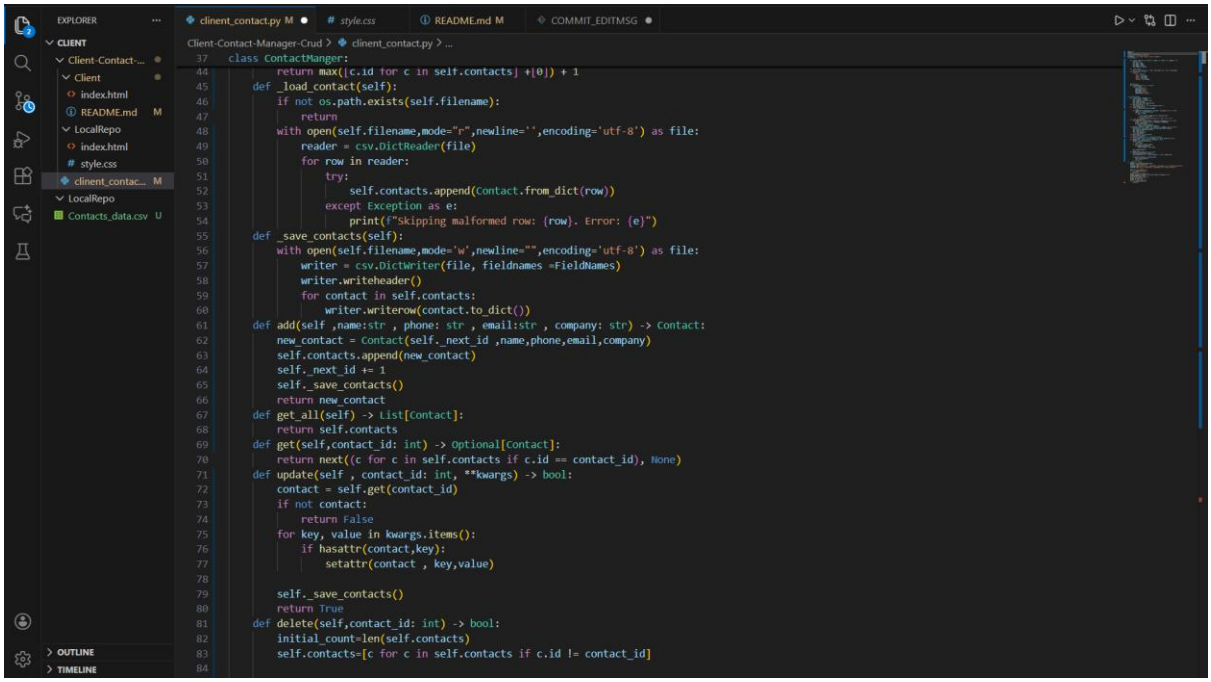
- **Technology:** Relies on the **Python Standard Library** (csv and os).
- **File:** The Contacts_data.csv file is read and overwritten (mode 'w') to maintain the single source of truth for all data.

# Screenshots / Results

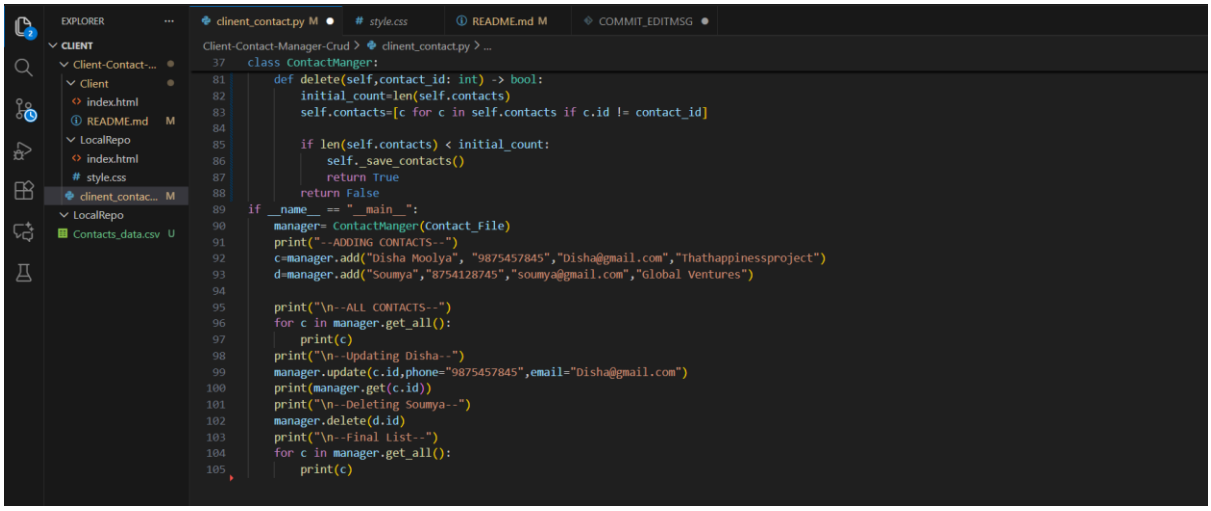
CODE -



```python
import csv
import os
from typing import List, Optional
Contact_File = "Contacts_data.csv"
FieldNames = ["id","name","phone","email","company"]
class Contact:

    def __init__(self,id: int, name: str, phone: str, email: str, company: str):
        self.id = id
        self.name = name
        self.email = email
        self.phone = phone
        self.company = company

    def __str__(self):
        return f"ID: {self.id:<3} | Name: {self.name:<20} | Phone: {self.phone}"
    def to_dict(self) -> dict:
        return {
            "id" : self.id,
            "name" : self.name,
            "phone": self.phone,
            "email": self.email,
            "company": self.company


        }
    @staticmethod
    def from_dict(data: dict) -> 'Contact':
        return Contact(
            id=int(data['id']),
            name=data['name'],
            phone=data['phone'],
            email=data['email'],
            comapny=data['company'],

        )
class ContactManger:
    def __init__(self, filename: str):
        self.filename = filename
        self.contacts: List[Contact]=[]
        self._load_contact()
        self._next_id = self._get_next_id()
```

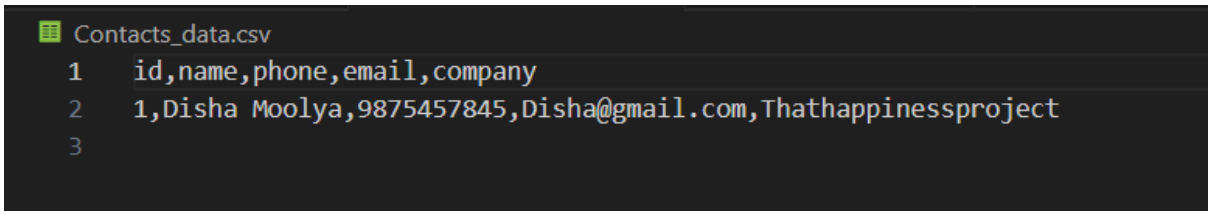


```python
class ContactManger:
        return max([c.id for c in self.contacts] +[0]) + 1
    def _load_contact(self):
        if not os.path.exists(self.filename):
            return
        with open(self.filename,mode="r",newline='',encoding='utf-8') as file:
            reader = csv.DictReader(file)
            for row in reader:
                try:
                    self.contacts.append(Contact.from_dict(row))
                except Exception as e:
                    print(f"Skipping malformed row: {row}. Error: {e}")
    def _save_contacts(self):
        with open(self.filename,mode='w',newline="",encoding='utf-8') as file:
            writer = csv.DictWriter(file, fieldnames =FieldNames)
            writer.writeheader()
            for contact in self.contacts:
                writer.writerow(contact.to_dict())
    def add(self ,name:str , phone: str , email:str , company: str) -> Contact:
        new_contact = Contact(self._next_id ,name,phone,email,company)
        self.contacts.append(new_contact)
        self._next_id += 1
        self._save_contacts()
        return new_contact
    def get_all(self) -> List[Contact]:
        return self.contacts
    def get(self,contact_id: int) -> Optional[Contact]:
        return next((c for c in self.contacts if c.id == contact_id), None)
    def update(self , contact_id: int, **kwargs) -> bool:
        contact = self.get(contact_id)
        if not contact:
            return False
        for key, value in kwargs.items():
            if hasattr(contact,key):
                setattr(contact , key,value)

        self._save_contacts()
        return True
    def delete(self,contact_id: int) -> bool:
        initial_count=len(self.contacts)
        self.contacts=[c for c in self.contacts if c.id != contact_id]
```


EXPLORER
CLIENT
Client-Contact-...
Client
index.html
README.md M
LocalRepo
index.html
style.css
clinent_contac... M
LocalRepo
Contacts_data.csv U
clinent_contact.py M
style.css
README.md M
COMMIT_EDITMSG
Client-Contact-Manager-Crud > clinent_contact.py > ...
class ContactManger:
def delete(self,contact_id: int) -> bool:
initial_count=len(self.contacts)
self.contacts=[c for c in self.contacts if c.id != contact_id]
if len(self.contacts) < initial_count:
self._save_contacts()
return True
return False
if __name__ == "__main__":
manager= ContactManger(Contact_File)
print("--ADDING CONTACTS--")
c=manager.add("Disha Moolya", "9875457845","Disha@gmail.com","Thathappinessproject")
d=manager.add("Soumya","8754128745","soumya@gmail.com","Global Ventures")
print("\n--ALL CONTACTS--")
for c in manager.get_all():
print(c)
print("\n--Updating Disha--")
manager.update(c.id,phone="9875457845",email="Disha@gmail.com")
print(manager.get(c.id))
print("\n--Deleting Soumya--")
manager.delete(d.id)
print("\n--Final List--")
for c in manager.get_all():
print(c)

RESULT -


--ADDING CONTACTS--
--ALL CONTACTS--
ID: 1 | Name: Disha Moolya | Phone: 9875457845
ID: 2 | Name: Soumya | Phone: 8754128745
--Updating Disha--
ID: 2 | Name: Soumya | Phone: 9875457845
--Deleting Soumya--
--Final List--
ID: 1 | Name: Disha Moolya | Phone: 9875457845


Contacts_data.csv
1 id,name,phone,email,company
2 1,Disha Moolya,9875457845,Disha@gmail.com,Thathappinessproject
3

# Testing Approach

The testing approach for the Client Contact CRUD Manager will primarily use **Unit Testing** and **Integration Testing** to verify functionality and data reliability.

---

**1. Unit Testing**

- **Focus:** Verifying individual methods of the Contact and ContactManager classes.
- **Key Tests:**
    - **CRUD Verification:** Ensure add, get, update, and delete methods perform their specific logic correctly (e.g., confirming delete returns True only when an ID is found).
    - **ID Management:** Verify that the _get_next_id method always generates a unique, correct sequential ID.
    - **Data Conversion:** Check that to_dict and from_dict correctly serialize and deserialize the Contact object.

**2. Integration Testing**

- **Focus:** Verifying the interaction between the application logic and the **Persistence Layer (CSV file)**.
- **Key Tests:**
    - **Persistence:** Confirm that data added during one test run is correctly **loaded** during the subsequent run (_load_contact).
    - **Data Integrity:** Verify that after any modification (add, update, delete), the contents of the **Contacts_data.csv** file accurately reflect the in-memory state.

**3. Scenario Testing**

- **Focus:** Testing application behavior under specific, critical conditions.
- **Key Tests:**
    - **Empty File Startup:** Verify the system handles initial startup gracefully when the Contacts_data.csv file does not exist.
    - **Non-existent IDs:** Test that get, update, and delete methods handle requests for non-existent IDs by returning None or False.

# Challenges Faced

The primary challenges encountered or inherent in this project's implementation involve constraints related to **data integrity, feature scope, and technology choice**:

- **Data Validation: Lack of input validation** (e.g., ensuring a phone number is numeric or an email is correctly formatted), which allows potentially bad data into the system.
- **Scalability:** Using a simple **CSV file** restricts the project's ability to scale efficiently for thousands of contacts, leading to slow performance for large datasets.
- **Concurrency/Multi-User Access:** The file-based system **does not support concurrent access**; simultaneous read/write operations from multiple users could lead to data corruption.
- **Error Handling:** The current code has **minimal error handling** for malformed CSV rows during load, relying on a basic try...except block which may skip critical data without sophisticated recovery.
- **Security: No built-in security** or access control mechanisms are present, as the project is designed for local, single-user access.

# Learnings & Key Takeaways

- **OOP for Structure:** Successfully used **Object-Oriented Programming (OOP)** to separate data (Contact class) from management logic (ContactManager), demonstrating clear code modularity.
- **Persistent Storage:** Implemented reliable **data persistence** using the Python **csv module**, ensuring data is automatically saved and loaded between sessions.
- **CRUD Implementation:** Gained practical experience implementing the **full set of CRUD** (Create, Read, Update, Delete) operations.
- **ID Management:** Learned to implement **automatic, unique ID generation** (_get_next_id) crucial for maintaining data integrity (primary key).
- **Standard Library Focus:** Developed a fully functional application relying **only on the Python Standard Library**, avoiding external dependencies.

# Future Enhancements

Potential future enhancements to the Client Contact CRUD Manager should focus on **improving data quality, user interaction, and feature robustness**:

- **Search and Filter:** Implement functions to **search** contacts by name, company, or email, and to **filter/sort** the list by specific criteria.
- **Data Validation:** Add **input validation** to ensure data integrity (e.g., checking for valid email formats, numeric phone numbers, and preventing blank required fields).
- **Advanced Persistence:** Migrate from CSV to a lightweight, dedicated database like **SQLite** to improve performance, data integrity, and support more complex queries.
- **User Interface (CLI):** Develop a more user-friendly **Command Line Interface (CLI)** that uses menus and prompts for interactive data entry instead of requiring direct function calls in the script.
- **Error Logging:** Implement detailed **logging** to record critical operations, warnings, and errors during file I/O.

# References

The project relies entirely on **built-in Python resources** and standard programming principles:

- **Python Documentation:** Official documentation for the Python language and its features.
- **CSV Module Documentation:** Official documentation detailing the usage of the built-in csv library for file reading and writing.
- **Object-Oriented Programming (OOP) Principles:** Standard concepts guiding the design of the Contact and ContactManager classes (e.g., encapsulation, modularity).
- **Software Design Principles:** General knowledge of **CRUD** (Create, Read, Update, Delete) architecture and **two-tier system architecture**.