```python
"""
Created by <REDACTED FOR ANONYMITY>
Created on 18/07/2025 at 15:35
"""
import csv
import time
from typing import List, Dict, Union, Optional


def handle_csv(filename: str, mode: str,
               data: Optional[List[Dict]] = None,
               fieldnames: Optional[List[str]] = None) -> Union[List[Dict], None]:
    """
    Generalized CSV handler for reading/writing CSV files.

    Args:
        filename: Name of the CSV file.
        mode: File mode ('r' for read, 'w' for write).
        data: Data to write (for write modes).
        fieldnames: Field names for writing.

    Returns:
        List of dictionaries when reading, None when writing.
    """
    with open(filename, mode, newline='') as csvfile:
        if mode == 'r':
            return list(csv.DictReader(csvfile))
        elif mode in ('w', 'a'):
            writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
            if mode == 'w':
                writer.writeheader()
            writer.writerows(data)
            return None
        return None

class Employee:
    def __init__(self, id: int, first_name: str, last_name: str) -> None:
        """
        A parent class for all employees.

        Args:
            id: A unique identifier for the employee.
            first_name: First name of the employee.
            last_name: Last name of the employee.
        """
        self.id = id
        self.first_name = first_name
        self.last_name = last_name


class Call:
    def __init__(self, id: int, status: str, time_elapsed: float = 0.0,
                 sat_score: float = 0.0, handler_id: int = 0) -> None:
        """
        Represents a call to an employee.

        Args:
            id: Unique ID for the call.
            status: Status from ["Successful", "Failed", "Pending"].
            time_elapsed: Time elapsed since call started (in seconds).
            sat_score: Satisfaction score of the call.
            handler_id: ID of the employee handling the call.
        """
        self.id = id
        self.status = status
        self.time_elapsed = time_elapsed
        self.sat_score = sat_score
        self.handler_id = handler_id


class Manager(Employee):
    def __init__(self, id: int, first_name: str, last_name: str, staff_list: List[int]) -> None:
        """
        Manager class inheriting from Employee.

        Args:
            id: Manager's unique ID.
            first_name: Manager's first name.
            last_name: Manager's last name.
            staff_list: List of staff IDs under this manager.
        """
        super().__init__(id, first_name, last_name)
        self.staff_list = staff_list
        print(f"New Manager Created: ID: {self.id}, First Name: {self.first_name}, "
              f"Last Name: {self.last_name}, Staff_List: {self.staff_list}")

    def add_staff(self, new_staff_id: int, first_name: str, last_name: str) -> None:
        """
        Add a new staff member to the team.

        Args:
            new_staff_id: ID of the new staff member.
            first_name: First name of the new staff.
            last_name: Last name of the new staff.
        """
        if new_staff_id not in self.staff_list:
            self.staff_list.append(new_staff_id)
            new_staff = {
                'staff_id': str(new_staff_id),
                'first_name': first_name,
                'last_name': last_name
            }
            print(f"Added staff with ID {new_staff_id} and name {first_name} {last_name}.")
        else:
            print(f"Staff with ID {new_staff_id} already exists.")

    def remove_staff(self, staff_id: int) -> None:
        """
        Remove a staff member from the team.

        Args:
            staff_id: ID of the staff member to remove.
        """
        if staff_id in self.staff_list:
            self.staff_list.remove(staff_id)
            staff_data = handle_csv('staff_details.csv', 'r')
            updated_data = [row for row in staff_data if row['staff_id'] != str(staff_id)]

            if updated_data:
                handle_csv('staff_details.csv', 'w', updated_data, list(updated_data[0].keys()))
            else:
                # If no data left, write empty file with headers
                handle_csv('staff_details.csv', 'w', [], ['staff_id', 'first_name', 'last_name'])
            print(f"Removed staff with ID {staff_id}.")
        else:
            print(f"Staff with ID {staff_id} not found.")

    def edit_staff_name(self, staff_id: int, new_first_name: str, new_last_name: str) -> None:
        """
        Edit a staff member's name.

        Args:
            staff_id: ID of the staff member.
            new_first_name: New first name.
            new_last_name: New last name.
        """
        staff_data = handle_csv('staff_details.csv', 'r')
        updated = False

        for row in staff_data:
            if row['staff_id'] == str(staff_id):
                row['first_name'] = new_first_name
                row['last_name'] = new_last_name
                updated = True
                break

        if updated:
            handle_csv('staff_details.csv', 'w', staff_data, list(staff_data[0].keys()))
            print(f"Staff {staff_id} name updated to {new_first_name} {new_last_name}")
        else:
            print(f"Staff with ID {staff_id} not found.")

    def view_staff_detail(self, staff_id: int) -> None:
        """
        View all details of a specific staff member.

        Args:
            staff_id: ID of the staff member to view.
        """
        staff_data = handle_csv('staff_details.csv', 'r')
        for row in staff_data:
            if row['staff_id'] == str(staff_id):
                print(",".join(f"{k}: {v}" for k, v in row.items()))
                return
        print(f"Staff with ID {staff_id} not found.")

    def view_staff_detail_selected(self, staff_id: int, fields_list: List[str]) -> None:
        """
        View selected details of a specific staff member.

        Args:
            staff_id: ID of the staff member to view.
            fields_list: List of fields to display.
        """
        staff_data = handle_csv('staff_details.csv', 'r')
        for row in staff_data:
            if row['staff_id'] == str(staff_id):
                selected = {k: v for k, v in row.items() if k in fields_list}
                if selected:
                    print("\n".join(f"{k}: {v}" for k, v in selected.items()))
                else:
                    print("None of the requested fields exist for this staff member.")
                return
        print(f"Staff with ID {staff_id} not found.")


class Staff(Employee):
    def __init__(self, id: int, first_name: str, last_name: str,
                 manager_id: int, calls_taken: int = 0, successful_calls: int = 0,
                 failed_calls: int = 0, target_successful_calls: int = 0,
                 working_time_elapsed: float = 0, avg_sat_score: float = 0,
                 status: str = "Out of Office") -> None:
        """
        Staff class inheriting from Employee.

        Args:
            id: Staff member's unique ID.
            first_name: Staff's first name.
            last_name: Staff's last name.
            manager_id: ID of the managing manager.
            calls_taken: Total calls taken.
            successful_calls: Number of successful calls.
            failed_calls: Number of failed calls.
            target_successful_calls: Target number of successful calls.
            working_time_elapsed: Total working time (seconds).
            avg_sat_score: Average satisfaction score.
            status: Current status from ['Free', 'On Call', 'Lunch', 'Out of Office'].
        """
        super().__init__(id, first_name, last_name)
        self.manager_id = manager_id
        self.calls_taken = calls_taken
        self.successful_calls = successful_calls
        self.failed_calls = failed_calls
        self.target_successful_calls = target_successful_calls
        self.working_time_elapsed = working_time_elapsed
        self.avg_sat_score = avg_sat_score
        self.status = status
        print(f"New Staff created with id: {self.id}, first name: {self.first_name}, last name: {self.last_name}")

    def accept_call(self, call: Call) -> None:
        """
        Accept an incoming call.

        Args:
            call: Call object to be accepted.
        """
        call.status = "In Progress"
        call.time_elapsed = time.time()
        call.handler_id = self.id
        self.status = "On Call"
        print(f"Call {call.id} accepted by {self.id}")

    def end_call(self, call: Call, user_sat_score:float) -> None:
        """
        End an ongoing call.

        Args:
            :param call: Call object to be ended.
            :param user_sat_score: The user's satisfaction score for the call
        """
        call.time_elapsed = time.time() - call.time_elapsed
        self.calls_taken += 1
        call.sat_score = user_sat_score
        if call.sat_score > 0.8:
            self.successful_calls += 1
            call.status = "Successful"
            else:
                self.failed_calls += 1
                call.status = "Failed"
        self.status = "Free"
        print(f"Call {call.id} ended by staff_id {self.id}")

    def see_call_history(self) -> None:
        """
        Display the call history for this staff member by reading from call records.

        Reads the call details CSV file and filters for calls handled by this staff member.
        Prints all call details (call_id, status, duration, satisfaction score, etc.)
        in a formatted manner if calls are found, otherwise displays a not found message.

        Returns:
            None: This method only prints output to console
        """
        call_data = handle_csv('call_details.csv', 'r')
        staff_calls = [row for row in call_data if row['handler_id'] == str(self.id)]

        if staff_calls:
            print("\nCall History:")
            for call in staff_calls:
                print(", ".join(f"{k}: {v}" for k, v in call.items()))
        else:
            print("No call history found for this staff member.")

    def start_workday(self) -> None:
        """
        Record the start time of a workday for this staff member.

        Sets the working_time_elapsed attribute to the current system time in seconds.
        This serves as the baseline for calculating total work duration when end_workday() is called.
        """
        self.working_time_elapsed = time.time()

    def end_workday(self) -> float:
        """
        Calculate and record the end of a workday, returning total duration worked.

        Computes the difference between current time and the start time recorded by
        start_workday(), then prints completion notification with timestamp.

        Returns:
            float: Total working duration in seconds
        """
        self.working_time_elapsed = time.time() - self.working_time_elapsed
        print(f"Staff ID {self.id} finished working at {time.gmtime(time.time())}")
        return self.working_time_elapsed
```