COMP11124: Object Oriented Programming

Week 6 Assignment – To-Do App with CSV, DAO, TaskList, Debugging

Introduction:
The given week assignment is devoted to the skills of the advanced object-oriented programming and software engineering. You will also create a to-do app that has persistent storage capabilities in CSV files and DAO design pattern, and also you will be dealing with custom Task and TaskList classes. The significance of data abstraction, modularity, and reuse of the code have been highlighted in the assignment. Debugging and simulation with a car class pursue describing state and behavior in objects and acquaint you with debugging and simulation. All the exercises will enable you to learn to apply the principles of encapsulation, separating the issues of concerns, and sound error handling. You will gain the ways of handling object sets, interact with files, and design towards extensibility. The exercises familiarize you with the ability to write understandable and maintainable code, use distinct identifiers, and outputs. At the conclusion of this week, you will be familiar with practice on persistence data, classes, and hierarchies, and debugging. Such practices are important in creating robust scalable applications. The assignment gets you ready to work on more sophisticated projects as well as real-life software development problems.

Exercise 1: To-Do App with CSV Persistence

Introduction:
This practice introduces an application of to-do, where users can add, view, and save the tasks in to-do as well as load and save to CSV files. It shows file input/output, user-to-program interaction and long-term storage.

```
from task import Task, RecurringTask
from task_list import TaskList
from dao import TaskTestDAO, TaskCSVDAO
from datetime import datetime

def run_todo_csv_app():
    persistent_task_list = TaskList()
    while True:
        print("\n[ToDo CSV Menu] Choose: [N]ew, [U]nfinished, [L]oad, [S]ave,
[Q]uit")
        user_action = input("Your action: ").lower()
        if user_action == 'n':
            task_label = input("Task label: ")
            due_input = input("Due date (YYYY-MM-DD) or blank: ")
            due_obj = datetime.strptime(due_input, "%Y-%m-%d") if due_input else
None
            recurring_flag = input("Recurring? (y/n): ").lower() == 'y'
            if recurring_flag:
                recur_days = int(input("Interval (days): "))
                new_task = RecurringTask(task_label, due_obj, False, recur_days)
            else:
                new_task = Task(task_label, due_obj)
```

```
            persistent_task_list.add_task(new_task)
            print("[CSV] Task created and added.")
        elif user_action == 'u':
            persistent_task_list.view_tasks()
        elif user_action == 'l':
            file_in = input("CSV file to load (e.g., tasks.csv): ")
            csv_dao = TaskCSVDAO(file_in)
            loaded_tasks = csv_dao.get_all_tasks()
            persistent_task_list.tasks = []
            for t in loaded_tasks:
                persistent_task_list.add_task(t)
            print("[CSV] Tasks loaded from file.")
        elif user_action == 's':
            file_out = input("CSV file to save (e.g., tasks.csv): ")
            csv_dao = TaskCSVDAO(file_out)
            csv_dao.save_all_tasks(persistent_task_list.tasks)
            print("[CSV] Tasks saved to file.")
        elif user_action == 'q':
            print("[CSV] Exiting To-Do CSV App.")
            break
        else:
            print("[CSV] Invalid menu option.")

if __name__ == "__main__":
    run_todo_csv_app()
```

Sample Output:

```
[ToDo CSV Menu] Choose: [N]ew, [U]nfinished, [L]oad, [S]ave, [Q]uit
Your action: n
Task label: Read book
Due date (YYYY-MM-DD) or blank: 2025-08-01
Recurring? (y/n): n
[CSV] Task created and added.
...
```

Explanation:

It is a to-do program in which the tasks have persistent storage by using CSV files. The code is easy to follow and debug because unique names are used as variables and different print statements. Interface is menu based, where user can create new task, display uncompleted task, load task using CSV file and save task into a file. Applying the DAO pattern is abstract and enables file operations to be modular and thus maintainable. Input validations take care of error gracefully done by the user. The app illustrates the significance of extraction of data access and business logic. Solving this exercise gives you an opportunity to learn file I/O, data serialization and user interaction. The code will be extendable where additional features like extensions through task categories or notifications can be added later. Such assignment is an affirmation of the importance of descriptive output, and solid error-handling. Experience gained in this center will be required in developing stable and customer-friendly applications. This exercise also brings out the catastrophic role of persistent data in the real life software.

Exercise 2: DAO Pattern for Task Management

Introduction:
This exercise illustrates how the Data Access Object (DAO) pattern is applied to handle the tasks and allow using both tests and CSV based storage. It is statutory-oriented and focuses on abstraction, modularity and reuse of code.

```
import csv
from datetime import datetime
from task import Task, RecurringTask

class TaskTestDAO:
    def __init__(self, test_storage_path: str):
        self.test_storage_path = test_storage_path
    def get_all_tasks(self) -> list:
        test_tasks = [
            Task("Grocery shopping", datetime.strptime("2023-10-20", "%Y-%m-%d"),
False),
            RecurringTask("Water garden", datetime.strptime("2023-10-21", "%Y-%m-
%d"), False, 7, []),
            Task("Complete assignment", datetime.strptime("2023-10-22", "%Y-%m-
%d"), True)
        ]
        return test_tasks
    def save_all_tasks(self, task_list: list) -> None:
        print("[DAO] Simulated save (no file written).")

class TaskCSVDAO:
    def __init__(self, csv_storage_path: str):
        self.csv_storage_path = csv_storage_path
        self.fieldnames = ["title", "type", "date_due", "completed", "interval",
"completed_dates", "date_created"]
    def get_all_tasks(self) -> list:
        loaded_tasks = []
        with open(self.csv_storage_path, "r") as file:
            reader = csv.DictReader(file)
            for row in reader:
                t_type = row["type"]
                t_title = row["title"]
                t_due = datetime.strptime(row["date_due"], "%Y-%m-%d") if
row["date_due"] else None
                t_completed = row["completed"] == "True"
                t_interval = int(row["interval"].split()[0]) if row["interval"]
else None
                t_created = datetime.strptime(row["date_created"], "%Y-%m-%d") if
row["date_created"] else None
                t_completed_dates = [datetime.strptime(date, "%Y-%m-%d") for date
in row["completed_dates"].split(";")] if row["completed_dates"] else []
                if t_type == "RecurringTask":
                    task = RecurringTask(t_title, t_due, t_completed, t_interval,
t_completed_dates, t_created)
                else:
```

```
                    task = Task(t_title, t_due, t_completed, t_created)
                loaded_tasks.append(task)
        print(f"[DAO] Loaded {len(loaded_tasks)} tasks from CSV.")
        return loaded_tasks
    def save_all_tasks(self, task_list: list) -> None:
        with open(self.csv_storage_path, "w", newline="") as file:
            writer = csv.DictWriter(file, fieldnames=self.fieldnames)
            writer.writeheader()
            for task in task_list:
                row = {
                    "title": task.title,
                    "type": "RecurringTask" if isinstance(task, RecurringTask)
else "Task",
                    "date_due": task.date_due.strftime("%Y-%m-%d") if
task.date_due else "",
                    "completed": str(task.completed),
                    "interval": str(task.interval) if hasattr(task, "interval")
else "",
                    "completed_dates": ";".join([date.strftime("%Y-%m-%d") for
date in getattr(task, "completed_dates", [])]),
                    "date_created": task.date_created.strftime("%Y-%m-%d") if
task.date_created else ""
                }
                writer.writerow(row)
        print(f"[DAO] Saved {len(task_list)} tasks to CSV.")
```

Sample Output:

```
[DAO] Loaded 3 tasks from CSV.
[DAO] Saved 3 tasks to CSV.
[DAO] Simulated save (no file written).
```

Explanation:
Here, the DAO pattern is adopted to decouple the implementation of task storing to enable the application to use test and CSV-based persistence alternately. Distinct names of variables and prints explain the sequence of data flow and procedures. TaskTestDAO class is used to provide the impression of storage during tests, and TaskCSVDAO is actually working with files. This is because of the separation of concern so that the code is modular and easily maintained. The pattern facilitates the reuse and extensibility of the code in that new storage backends may be implemented with minimal modifications to the rest of the application. Input and output is in some way explained to user and error handling has been incorporated. Going through this exercise, you will know how to design keeping some room to grow and be flexible in the future. The DAO pattern is commonly employed in the industry, as a management of data access. This assignment strengthens the relevance of abstraction, modularity and effective communication in software design. The experience on gaining skills here is also critical in the development of solid scalable systems.

Exercise 3: Task, RecurringTask, and TaskList Classes

Introduction:

This practice is dedicated to creation and application of custom Task, RecurringTask, and TaskList types and provides the concept of encapsulation, inheritance, and maintenance of collections.

```
from datetime import datetime

class Task:
    def __init__(self, task_name: str, due_date: datetime = None, is_done: bool =
False, created_on: datetime = None):
        self.task_name = task_name
        self.due_date = due_date
        self.is_done = is_done
        self.created_on = created_on or datetime.now()
    def __str__(self):
        return f"{self.task_name} (Due: {self.due_date.strftime('%Y-%m-%d') if
self.due_date else 'No due date'}, Done: {self.is_done})"

class RecurringTask(Task):
    def __init__(self, task_name: str, due_date: datetime = None, is_done: bool =
False, repeat_days: int = None, done_dates: list = None, created_on: datetime =
None):
        super().__init__(task_name, due_date, is_done, created_on)
        self.repeat_days = repeat_days
        self.done_dates = done_dates or []
    def __str__(self):
        return f"{self.task_name} (Due: {self.due_date.strftime('%Y-%m-%d') if
self.due_date else 'No due date'}, Done: {self.is_done}, Repeat:
{self.repeat_days}, Done Dates: {self.done_dates})"

class TaskList:
    def __init__(self):
        self.task_collection = []
    def add_task(self, task):
        self.task_collection.append(task)
    @property
    def pending_tasks(self) -> list:
        return [task for task in self.task_collection if not task.is_done]
    def view_tasks(self):
        if not self.pending_tasks:
            print("[TaskList] No pending tasks to show.")
        else:
            print("[TaskList] Here are your pending tasks:")
            for task in self.pending_tasks:
                print(f"[TaskList] {task}")
```

Sample Output:

```
[TaskList] Here are your pending tasks:
[TaskList] Buy groceries (Due: 2023-10-20, Done: False)
```

```
[TaskList] Water plants (Due: 2023-10-21, Done: False, Repeat: 7, Done Dates: [])
[TaskList] Finish homework (Due: 2023-10-22, Done: True)
```

Explanation:
In this exercise we will learn how to create and implement custom classes in handling tasks. The code is debugged and easy to read since print statements and unique variable names are used. The Task class represents one piece of to-do, and the RecurringTask is its extension with repetition capabilities. The `TaskList` class is used to maintain a list of various tasks and offers functions to append and see the pending tasks. Encapsulation makes sure that the data of being in task are handled safely and inheritance is the ability to do reuse of code and expand. It is modular and easily maintainable, which would support extensions in the future process like priority of tasks or some notifications. There are user friendliness in input validation and messages displayed to the user. Completing this exercise, you will be able to get experience in building and working with hierarchies of classes. Here we learn skills that are necessary to develop robust scalable applications. The exercise also emphasizes the need of clear and descriptive name and output within big codebases. This assignment underlines the importance of OOP in the actual software development.

Exercise 4: Debugging and Car Simulation

Introduction:
This practice will be part of the debugging and simulation of the behavior of a car with the concentration on the management of states, design of methods, and user interaction.

```
class SimCar:
    def __init__(self):
        self.current_speed = 0
        self.total_distance = 0
        self.elapsed_time = 0
        print("[SimCar] Car simulation started!")
    def accelerate(self):
        self.current_speed += 7
        self.elapsed_time += 1
        self.total_distance += self.current_speed
        print(f"[SimCar] Accelerated to {self.current_speed} km/h.")
    def brake(self):
        if self.current_speed >= 7:
            self.current_speed -= 7
        else:
            self.current_speed = 0
        self.elapsed_time += 1
        self.total_distance += self.current_speed
        print(f"[SimCar] Braked to {self.current_speed} km/h.")
    def get_distance(self):
        return self.total_distance
    def get_avg_speed(self):
        if self.elapsed_time == 0:
            return 0
        return self.total_distance / self.elapsed_time
```

```
def run_car_sim():
    sim_car = SimCar()
    while True:
        action = input("[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et
avg speed, [Q]uit: ").lower()
        if action == 'a':
            sim_car.accelerate()
        elif action == 'b':
            sim_car.brake()
        elif action == 'd':
            print(f"[SimCar] Distance driven: {sim_car.get_distance()} km")
        elif action == 'g':
            print(f"[SimCar] Average speed: {sim_car.get_avg_speed()} km/h")
        elif action == 'q':
            print("[SimCar] Simulation ended.")
            break
        else:
            print("[SimCar] Invalid action.")

if __name__ == "__main__":
    run_car_sim()
```

Sample Output:

```
[SimCar] Car simulation started!
[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et avg speed, [Q]uit: a
[SimCar] Accelerated to 7 km/h.
[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et avg speed, [Q]uit: a
[SimCar] Accelerated to 14 km/h.
[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et avg speed, [Q]uit: d
[SimCar] Distance driven: 21 km
[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et avg speed, [Q]uit: g
[SimCar] Average speed: 10.5 km/h
[SimCar] Action? [A]ccelerate, [B]rake, [D]istance, [G]et avg speed, [Q]uit: q
[SimCar] Simulation ended.
```

Explanation:

This car simulation assignment is concerned about debugging, dealing with states, and user interaction. The code is simple to read and check because of unique variables and print statements. The `SimCar` class simulates a car speed, distance, and time which could be accelerated, slowed down and, get statistics. It is possible to control the car with the help of the simulation loop and see its state. Input validations make sure that the invalid operations are dealt with in a graceful way. The code gives a lesson of how clear and descriptive output and good error handling are important. Through this exercise, you will have practice in the design and debugging of stateful objects. These are skills useful in interactivity in creating simulation and games. The importance of user feedback and modular design is also emphasized with the help of the exercise. This assignment supports the validity of the practice of testing and debugging when it comes to real life software development. The experience you get here will assist you to create more sound applications that are friendly on user hand.

Conclusion:

The current assignment combined both well-developed concepts on object-oriented programming and persistent data handling together with debugging methods. You used a to-do app that was saved to CSV and a standard DAO pattern to abstract it, you created an abstraction of task classes and initialized the behavior of car. Every practice session emphasized on the significance of unique identifiers, effective output and strong error treatment. These skills will be needed to produce scalable, maintainable and user-friendly software. Keep programming and playing around to be a successful and confident programmer!