COMP11124: Object Oriented Programming

Week 5 Assignment – Portfolio, Enhanced To-Do, Vehicle Hierarchy, UML

Introduction:
This week assignment is an in-depth study of a concept of object oriented programming and its practical implementation. You will create a portfolio and expand a to-do app containing support of users and repeating tasks, create an advanced to-do application with more functionality, and develop a hierarchy of a vehicle to evidence the concept of inheritance and polymorphism. You will also draw a UML diagram to draw up mapping of your structures and relations of classes. The exercises are all made to strengthen your knowledge about classes and inheritance, encapsulation, code structure. You will get a feel of handling input of users, handling groups of objects and dealing with events. The assignment too focuses on code readability, modularity and testing. At the end of this week, you will be fully aware of advanced OOP methodologies and be ready to plug them into practice. The skills accumulated in it are preliminary to more significant software projects and future coursework.

Exercise 1: Portfolio To-Do Application

Introduction:
Such exercise incorporates a portfolio to-do application with support of users, owners, normal and routine tasks, and task list. It illustrates design of classes, user interaction and management of tasks.

```
# main.py
import datetime
from users import Owner
from tasks import Task, RecurringTask, TaskList

def main():
    print("=== Portfolio To-Do App Booting Up ===")
    user_fullname = input("Please enter your full name: ")
    user_email = input("Please enter your email address: ")
    portfolio_owner = Owner(user_fullname, user_email)
    portfolio_tasklist = TaskList(portfolio_owner)
    print(f"\nHello, {portfolio_owner}! Your portfolio to-do list is ready.")
    while True:
        print("\n--- Portfolio To-Do Menu ---")
        print("1. Add New Portfolio Task")
        print("2. Delete Portfolio Task")
        print("3. Show All Portfolio Tasks")
        print("4. Show Overdue Portfolio Tasks")
        print("5. Complete a Portfolio Task")
        print("6. Exit Portfolio App")
        menu_option = input("Choose an option (1-6): ")
        if menu_option == "1":
            task_name = input("Task name: ")
            due_str = input("Due date (YYYY-MM-DD): ")
```

```
            due_dt = datetime.datetime.strptime(due_str, "%Y-%m-%d")
            recurring = input("Recurring task? (y/n): ").lower()
            if recurring == "y":
                recur_days = int(input("Recurrence interval (days): "))
                recur_delta = datetime.timedelta(days=recur_days)
                new_task = RecurringTask(task_name, due_dt, recur_delta)
            else:
                task_desc = input("Task details (optional): ")
                new_task = Task(task_name, due_dt, task_desc)
            portfolio_tasklist.add_task(new_task)
            print("Portfolio task successfully added!")
        elif menu_option == "2":
            portfolio_tasklist.view_tasks()
            if portfolio_tasklist.tasks:
                try:
                    del_idx = int(input("Task number to delete: ")) - 1
                    portfolio_tasklist.remove_task(del_idx)
                    print("Portfolio task deleted!")
                except (ValueError, IndexError):
                    print("Invalid task number for deletion!")
        elif menu_option == "3":
            portfolio_tasklist.view_tasks()
        elif menu_option == "4":
            portfolio_tasklist.view_overdue_tasks()
        elif menu_option == "5":
            portfolio_tasklist.view_tasks()
            if portfolio_tasklist.tasks:
                try:
                    comp_idx = int(input("Task number to complete: ")) - 1
                    task = portfolio_tasklist.get_task(comp_idx)
                    task.mark_completed()
                    print("Portfolio task marked as completed!")
                except (ValueError, IndexError):
                    print("Invalid task number for completion!")
        elif menu_option == "6":
            print(f"Goodbye, {portfolio_owner.name}! Portfolio session ended.")
            break
        else:
            print("Invalid menu option. Please try again.")

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

Sample Output:

```
=== Portfolio To-Do App Booting Up ===
Please enter your full name: Alice Smith
Please enter your email address: alice@example.com

Hello, Owner: Alice Smith (alice@example.com)! Your portfolio to-do list is ready.

--- Portfolio To-Do Menu ---
```

```
1. Add New Portfolio Task
2. Delete Portfolio Task
3. Show All Portfolio Tasks
4. Show Overdue Portfolio Tasks
5. Complete a Portfolio Task
6. Exit Portfolio App
Choose an option (1-6): 1
Task name: Write blog post
Due date (YYYY-MM-DD): 2025-07-25
Recurring task? (y/n): n
Task details (optional): Write about OOP
Portfolio task successfully added!
...
```

Explanation:
The scope of this portfolio to-do application includes assisting its user to handle regular as well as recurring tasks, where a task has an owner. It illustrates the use of inheritance in the type of tasks in the code and encapsulation in user and task data. menu-driven interface is used to add, delete, view and complete tasks, and see those things that are overdue. This backdating mechanism is in effect with recurring tasks, whose due dates are automatically updated as the task is done, hence demonstrating the overriding effect of method. The application also decouples concerns because the code is modular since users, tasks, and task list use different classes. There is validation of the inputs to ensure that the errors by the user are accommodated gracefully. The design can be scaled down, which makes additional implementation of such features as notifications or task types possible in the future. The reason behind this exercise is stressing the need of object-oriented design, interaction with user and good error handling. In completing this task you acquire some real-life experience in OOP applications construction. Skills received here can be directly used in bigger projects and professional software developing.

Exercise 2: Enhanced To-Do Application

Introduction:
This assignment is the extension of the to-do application with enhanced tasks management, recurring tasks, and overdue tasks tracking, with the emphasis on code structure and usability.

```python
# main.py
import datetime
from tasks import Task, RecurringTask, TaskList

def main():
    user_nickname = input("Enter your preferred name: ")
    enhanced_list = TaskList(user_nickname)
    while True:
        print("\n--- Enhanced To-Do Menu ---")
        print("1. Add Enhanced Task")
        print("2. Remove Enhanced Task")
        print("3. List Enhanced Tasks")
        print("4. List Overdue Enhanced Tasks")
        print("5. Complete Enhanced Task")
```

```
        print("6. Exit Enhanced App")
        menu_pick = input("Select an option (1-6): ")
        if menu_pick == "1":
            task_label = input("Task label: ")
            due_str = input("Due date (YYYY-MM-DD): ")
            due_dt = datetime.datetime.strptime(due_str, "%Y-%m-%d")
            is_recurring = input("Recurring? (y/n): ").lower()
            if is_recurring == "y":
                recur_days = int(input("Recurrence (days): "))
                recur_delta = datetime.timedelta(days=recur_days)
                task_obj = RecurringTask(task_label, due_dt, recur_delta)
            else:
                desc = input("Description (optional): ")
                task_obj = Task(task_label, due_dt, desc)
            enhanced_list.add_task(task_obj)
            print("Enhanced task added!")
        elif menu_pick == "2":
            enhanced_list.view_tasks()
            if enhanced_list.tasks:
                del_idx = int(input("Task number to remove: ")) - 1
                enhanced_list.remove_task(del_idx)
                print("Enhanced task removed!")
        elif menu_pick == "3":
            print("\n--- Your Enhanced Tasks ---")
            enhanced_list.view_tasks()
        elif menu_pick == "4":
            print("\n--- Overdue Enhanced Tasks ---")
            enhanced_list.view_overdue_tasks()
        elif menu_pick == "5":
            enhanced_list.view_tasks()
            if enhanced_list.tasks:
                comp_idx = int(input("Task number to complete: ")) - 1
                task = enhanced_list.get_task(comp_idx)
                task.mark_completed()
                print("Enhanced task marked as completed!")
        elif menu_pick == "6":
            print("Exiting Enhanced To-Do App. See you soon!")
            break
        else:
            print("Invalid menu selection. Try again.")

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

Sample Output:

```
Enter your preferred name: Bob

--- Enhanced To-Do Menu ---
1. Add Enhanced Task
2. Remove Enhanced Task
3. List Enhanced Tasks
```

```
4. List Overdue Enhanced Tasks
5. Complete Enhanced Task
6. Exit Enhanced App
Select an option (1-6): 1
Task label: Pay rent
Due date (YYYY-MM-DD): 2025-07-20
Recurring? (y/n): y
Recurrence (days): 30
Enhanced task added!
...
```

Explanation:
The improved to-do application is an extension of the portfolio application, whereby it includes more comfortable features and feedback. The code is easy to read and debug as names of variables used are unique, and print statements are used. The app has the features of repeating links, tracking of the overdue tasks, and a modular task management. Its menu is also intuitive and everything done receives a unique message of confirmation. They are broken up into the code structure that prevents user and task logic in favor of maintainability. Input controls are supposed to make the app graceful in the event of an error, and to automatically reschedule long-running activities. The exercise shows the significance of usability, clearness of code and handling of errors. In solving this activity, you advance your skills in creating scalable applications that focus on users. The lessons taught here are useful in developing professional softwares. Another important lesson of the exercise is the importance of user-friendly communication with the help of special prompts and confirmations.

Exercise 3: Vehicle Class Hierarchy

Introduction:
In this exercise, inheritance, polymorphism, and multiple inheritance are illustrated using a vehicle hierarchy in which car, plane, and other special classes are defined.

```
import datetime

class Vehicle:
    def __init__(self, paint: str, mass: float, velocity_limit: float, **kwargs):
        self.paint = paint
        self.mass = mass
        self.velocity_limit = velocity_limit
        self.range_limit = kwargs.get('range_limit', None)
        self.seating = kwargs.get('seating', None)
    def move(self, velocity: float) -> None:
        print(f"[Vehicle] Now moving at {velocity} km/h!")

class Car(Vehicle):
    def __init__(self, paint: str, mass: float, velocity_limit: float, body_type:
str, **kwargs):
        super().__init__(paint, mass, velocity_limit, **kwargs)
        self.body_type = body_type
    def move(self, velocity: float) -> None:
```

```
        print(f"[Car] Cruising at {velocity} km/h!")

class ElectricCar(Car):
    def __init__(self, paint: str, mass: float, velocity_limit: float, body_type:
str, battery_kwh: float, **kwargs):
        super().__init__(paint, mass, velocity_limit, body_type, **kwargs)
        self.battery_kwh = battery_kwh
    def move(self, velocity: float) -> None:
        range_info = f" (max range: {self.range_limit} km)" if self.range_limit
else ""
        print(f"[ElectricCar] Gliding at {velocity} km/h{range_info}")

class PetrolCar(Car):
    def __init__(self, paint: str, mass: float, velocity_limit: float, body_type:
str, tank_liters: float, **kwargs):
        super().__init__(paint, mass, velocity_limit, body_type, **kwargs)
        self.tank_liters = tank_liters
    def move(self, velocity: float) -> None:
        range_info = f" (max range: {self.range_limit} km)" if self.range_limit
else ""
        print(f"[PetrolCar] Roaring at {velocity} km/h{range_info}")

class Plane(Vehicle):
    def __init__(self, paint: str, mass: float, velocity_limit: float, wingspan_m:
float, **kwargs):
        super().__init__(paint, mass, velocity_limit, **kwargs)
        self.wingspan_m = wingspan_m
    def move(self, velocity: float) -> None:
        print(f"[Plane] Flying at {velocity} km/h!")

class PropellerPlane(Plane):
    def __init__(self, paint: str, mass: float, velocity_limit: float, wingspan_m:
float, prop_diameter: float, **kwargs):
        super().__init__(paint, mass, velocity_limit, wingspan_m, **kwargs)
        self.prop_diameter = prop_diameter
    def move(self, velocity: float) -> None:
        print(f"[PropellerPlane] Spinning at {velocity} km/h!")

class JetPlane(Plane):
    def __init__(self, paint: str, mass: float, velocity_limit: float, wingspan_m:
float, thrust_kn: float, **kwargs):
        super().__init__(paint, mass, velocity_limit, wingspan_m, **kwargs)
        self.thrust_kn = thrust_kn
    def move(self, velocity: float) -> None:
        print(f"[JetPlane] Zooming at {velocity} km/h!")

class FlyingCar(Car, Plane):
    def __init__(self, paint: str, mass: float, velocity_limit: float, body_type:
str, wingspan_m: float, **kwargs):
        super().__init__(paint=paint, mass=mass, velocity_limit=velocity_limit,
body_type=body_type, wingspan_m=wingspan_m, **kwargs)
    def move(self, velocity: float) -> None:
        print(f"[FlyingCar] Hybrid driving/flying at {velocity} km/h!")
```

```
# Testing the classes
if __name__ == "__main__":
    v1 = Vehicle("silver", 1200, 180)
    v1.move(90)
    c1 = Car("black", 1400, 220, "Sedan")
    c1.move(120)
    e1 = ElectricCar("white", 1300, 200, "Coupe", 85, range_limit=400, seating=4)
    e1.move(110)
    p1 = PetrolCar("red", 1500, 210, "SUV", 60, range_limit=550)
    p1.move(130)
    pl1 = Plane("blue", 40000, 850, 28)
    pl1.move(700)
    pr1 = PropellerPlane("yellow", 35000, 600, 22, 3.5)
    pr1.move(400)
    j1 = JetPlane("white", 50000, 950, 35, 120)
    j1.move(900)
    fc1 = FlyingCar("green", 1100, 180, "Convertible", 15, seating=2)
    fc1.move(160)
    print(f"Seats: {fc1.seating}, Wingspan: {fc1.wingspan_m}, Body:
{fc1.body_type}")
```

Sample Output:

```
[Vehicle] Now moving at 90 km/h!
[Car] Cruising at 120 km/h!
[ElectricCar] Gliding at 110 km/h (max range: 400 km)
[PetrolCar] Roaring at 130 km/h (max range: 550 km)
[Plane] Flying at 700 km/h!
[PropellerPlane] Spinning at 400 km/h!
[JetPlane] Zooming at 900 km/h!
[FlyingCar] Hybrid driving/flying at 160 km/h!
Seats: 2, Wingspan: 15, Body: Convertible
```

Explanation:

The exercise on languages shows that the order of vehicle class can be modeled by the use of inheritance and polymorphism to reflect many related objects. In order to make classes more interpretable and understand their role in the program, each of them has its own variable names and direct print statements. A vehicle is the base class, and cars and planes inherit it and in turn are specialized to be electric cars, petrol cars, propellor cars, and jet cars. The multple inheritance can be observed in the `FlyingCar` class. To make each subclass express its distinct behavior, each overrides the `move` method with a different way of doing the same task and the test code located at the bottom illustrates the working action of each type of vehicle. It is simple to expand and alter this structure hence it is appropriate to intricate systems. The code is readable, the code is modular and we see all the best practices in OOP. Through the course of this exercise, you have learned to come up with adaptable, re-usable lines of classes. Unique identifiers and outputs can be used which makes the code easy to maintain and debug. The task also emphasizes on descriptive and clear naming and output in a large code base.

Exercise 4: UML Diagram

Introduction:
This exercise requires you to create a UML class diagram (see `class.drawio`) that visually represents the relationships between your classes in the to-do and vehicle hierarchy applications.

UML Diagram:

- The UML diagram should show all classes, their attributes, methods, and relationships (inheritance, association, etc.).
- Use a tool like draw.io to create the diagram and export it as an image or PDF for submission.

Sample Output:
(UML diagrams are visual; see the attached `class.drawio` file for the actual diagram.)

Explanation:
UML class diagram is tool that can effectively show the format of your code. It depicts the association between classes, which attributes and methods they contain and how inheritance and associations are structured. When you design a UML diagram, you can easily share the design with other people and identify pitfalls to improve. Such exercise exorcizes the need to have a plan and paper trail of your software architecture. It also assists you in getting the high level picture of your application which subsequently makes it simpler to maintain as well as expand. Software design UML diagrams are common in both the industry and academia. They are particularly useful where projects are numerous and have an extensive number of classes and relationships. When you practice UML, you would not only acquire skills that are useful in dealing with technical interviews but also in the real world development. The task will also stimulate you to consider critically your class design and its clarity.

Conclusion:

The assignment this week along with the concepts of advanced object oriented programming, practical application development and visualization of software design came together. You modeled to-do list applications, you made applications run overdue and recurring tasks, and you created a complicated vehicle hierarchy. The UML diagram strengthened your design and class relationship. These assignments have readied you up to bigger projects and practical problems involved with software engineering. Continue developing such skills and become a solid and confident developer!