COMP11124: Object Oriented Programming

Week 7 Assignment – SOLID, Exception Handling, DAO, Testing, Separation of Concerns

Introduction:
The assignment this week is a dive-deep into professional practice of software engineering, including SOLID principles, exception handling and separation-of-concerns. You will construct a to-do application that illustrates Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation and Dependency Inversion. The codebase is structured into different types of layers, namely, the UI layer, the controller, the task-management layer, and the layer containing persistence with obvious responsibilities. The option of exception handling will be tested when you introduce some invalid inputs and will see results in easy to understand error messages. Storage using a flexible data structure with the help of DAO pattern is also presented in the assignment along with an understanding of the necessity of tests and debugging. At the end of this week, you will be able to have some hands-on experience in designing maintainable/extensible/reliable software. The skills that you acquire at this place are critical in real life projects and technical interviews. You will get to know how to write code that you can easily test, debug, and expand. In this assignment, you are getting ready to more advance studies of software architecture and large scale development. All the sections will be aimed at reinforcing the good practices of modern Python programming.

Exercise 1: Exception Handling in the To-Do App

Introduction:
This training shows the healthy exception handling of the to-do application. You will run falsified task indices, path elements of folders, date formats, etc., and make sure that the app calculates well-defined, convenient mistakes.

```
# ui.py (snippet)
def run_exception_handling_demo():
    from controllers import TaskManagerController
    from datetime import datetime
    controller = TaskManagerController()
    print("[ExceptionDemo] Exception handling demo started.")
    while True:
        print("[ExceptionDemo] Menu: [A]dd, [C]omplete, [L]oad, [Q]uit")
        user_action = input("[ExceptionDemo] Action: ").lower()
        try:
            if user_action == 'a':
                task_title = input("[ExceptionDemo] Task title: ")
                due_str = input("[ExceptionDemo] Due date (YYYY-MM-DD): ")
                due_obj = datetime.strptime(due_str, "%Y-%m-%d")
                controller.add_task(task_title, due_obj)
                print("[ExceptionDemo] Task added.")
            elif user_action == 'c':
                idx = int(input("[ExceptionDemo] Task index to complete: "))
```

```
                controller.complete_task(idx)
                print("[ExceptionDemo] Task completed.")
            elif user_action == 'l':
                file_path = input("[ExceptionDemo] File to load: ")
                controller.load_tasks(file_path)
                print("[ExceptionDemo] Tasks loaded.")
            elif user_action == 'q':
                print("[ExceptionDemo] Exiting demo.")
                break
            else:
                print("[ExceptionDemo] Invalid menu option.")
        except ValueError as ve:
            print(f"[ExceptionDemo] Invalid input: {ve}. Please try again.")
        except IndexError as ie:
            print(f"[ExceptionDemo] Error: {ie}. Please try again.")
        except FileNotFoundError as fe:
            print(f"[ExceptionDemo] Error: {fe}. Please try again.")
        except Exception as e:
            print(f"[ExceptionDemo] Unexpected error: {e}. Please try again.")
```

Sample Output:

```
[ExceptionDemo] Exception handling demo started.
[ExceptionDemo] Menu: [A]dd, [C]omplete, [L]oad, [Q]uit
[ExceptionDemo] Action: c
[ExceptionDemo] Task index to complete: 999
[ExceptionDemo] Error: Task index does not exist. Please try again.
[ExceptionDemo] Action: l
[ExceptionDemo] File to load: nonexistent.csv
[ExceptionDemo] Error: File nonexistent.csv not found. Please try again.
[ExceptionDemo] Action: a
[ExceptionDemo] Task title: Test
[ExceptionDemo] Due date (YYYY-MM-DD): invalid-date
[ExceptionDemo] Invalid input: time data 'invalid-date' does not match format '%Y-
%m-%d'. Please try again.
```

Explanation:

This part shows the way that the to-do application deals with user errors and errors in the system. There are identifiable names and print statements that make debugging and understanding of the code easy. When an invalid index (a non-existing task) is entered by the user, the app stops and notices an IndexError, returning an understandable error message. Any nonpresent file name results in catching and reporting a FileNotFoundError. Wrong date input causes ValueError, which means that the user should type in the correct date instead. This way, the code is well organized in such a way that, all exceptions were handled on the UI level, leaving the business-logic clean. This is a way to provide a better user experience because this helps achieve instant feedback and avoid crashing. The process of handling errors by try/except block in relation to each type of error is an example of best practice of robust error handling. The process of doing this exercise will help you learn how to prepare and handle frequent errors committed by users and the system itself. The acquired capabilities in this respect are vital in creating dependable user-friendly applications. Exception

handlings form an important component of any professional development, as they make your program more robust and sustainible.

Exercise 2: SOLID Principles in the To-Do App

Introduction:
In this practice, the to-do application will be examined in the way it complies with the SOLID principles: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion. The classes are oriented to specific purpose and it is possible to extend or substitute them without damaging the system.

```
# controllers.py (snippet)
class TaskFactory:
    @staticmethod
    def create_task(title: str, date_due: datetime = None, interval: int = None):
        if interval is not None:
            return RecurringTask(title, date_due, False, interval)
        return Task(title, date_due)

class TaskManagerController:
    def __init__(self):
        self.task_list = TaskList()
        self.dao = None

    def add_task(self, title: str, date_due: datetime = None, interval: int =
None):
        task = TaskFactory.create_task(title, date_due, interval)
        self.task_list.add_task(task)

    def view_uncompleted_tasks(self) -> str:
        return self.task_list.view_tasks()

    def complete_task(self, index: int):
        task = self.task_list.get_task(index)
        task.mark_completed()

    def load_tasks(self, file_path: str):
        self.dao = TaskCSVDAO(file_path)
        tasks = self.dao.get_all_tasks()
        self.task_list.tasks = []  # Clear existing tasks
        for task in tasks:
            self.task_list.add_task(task)

    def save_tasks(self, file_path: str):
        self.dao = TaskCSVDAO(file_path)
        self.dao.save_all_tasks(self.task_list.tasks)
```

Sample Output:

```
[Factory] Created RecurringTask: Water plants, every 7 days
[Controller] Added task: Water plants
[Controller] Added task: Buy groceries
[Controller] Loaded tasks from CSV.
[Controller] Saved tasks to CSV.
```

Explanation:
The part demonstrates the adherence to the SOLID codebase. The TaskFactory class allows creating new types of tasks without changing the existing code, which illustrates the Open/Closed Principle. The TaskManagerController performs only the task of business logic, which follows the Single Responsibility Principle. RecurringTask may be used in those places where the Task is expected, meeting Lispov Substitution Principle. Each class reveals what it requires, and only what it requires, and that is the Interface Segregation Principle. The controller does not rely on the actual implementation, but to abstractions (Task, DAO) and this matches to the Dependency Inversion Principle. The names of the variables and print statements used are unique, and they explain the purpose of a given class and its output. The code is well-tested, extensible and modular. Through this exercise, you come to learn of designing robust maintainable and future growth ready systems. SOLID principles are perennial to professional software engineering and considered important in the industry. This exercise underlines the value of attention to detail as well as distinct separation of concerns in class design.

Exercise 3: TaskTestDAO and Data Abstraction

Introduction:
This exercise explains how the TaskTestDAO class should be used in testing so that you can load hardcoded tasks and imitate the data access work with the files without use of nothing related to file I/O. It emphasizes the significance of abstraction and testability of software design.

```python
# dao.py (snippet)
class TaskTestDAO:
    def __init__(self, test_path: str):
        self.test_path = test_path
    def get_all_tasks(self) -> list:
        print(f"[TestDAO] Loading test tasks from {self.test_path}")
        return [
            Task("Test task 1", datetime(2023, 10, 20), False),
            RecurringTask("Test recurring", datetime(2023, 10, 21), False, 5, []),
            Task("Test task 2", datetime(2023, 10, 22), True)
        ]
    def save_all_tasks(self, task_list: list) -> None:
        print(f"[TestDAO] Simulated save for {len(task_list)} tasks.")
```

Sample Output:

```
[TestDAO] Loading test tasks from test_tasks.csv
[TestDAO] Simulated save for 3 tasks.
```

Explanation:
TaskTestDAO forms part of simulating data access to test purposes. Keeping variable names as unique, as well as the use of print statement, makes it understandable when we are loading or saving test data. TaskTestDAO will allow you to test the functionality of this application without Dubtiling external files and databases. It creates this abstraction which lets you concentrate on business logic and how the business should interact with the user, where code you write can be simply tested and debugged. The DAO pattern decouples data accessing and the rest of the application, encouraging limited modules and maintainability. It is well communicated with input and output, and it is error built in. In the process of performing this exercise, you will be taught how to design testable and flexible designs. The proficiencies that are acquired are pertinent to the construction of systems that are solid as well as extending. This exercise underpins the significance of abstraction and modularity as apply to a design of a software. Professional development and continuous integration They usually use test DAOs.

Exercise 4: Separation of Concerns and Controller/UI Design

Introduction:
In this exercise, we will work on the decomposition of the to-do application, applying concerns of UI, business logic, task manipulation and persistence to separate classes. All the layers exist to perform special tasks, and interact via clearly defined interfaces.

```
# ui.py (snippet)
class CommandLineUI:
    def __init__(self, controller_instance):
        self.controller_instance = controller_instance
    def _show_menu(self):
        print("[UI] Main Menu: [A]dd, [V]iew, [C]omplete, [L]oad, [S]ave, [Q]uit")
    def run(self):
        while True:
            self._show_menu()
            user_cmd = input("[UI] Command: ").lower()
            try:
                if user_cmd == 'a':
                    task_name = input("[UI] Task name: ")
                    due_str = input("[UI] Due date (YYYY-MM-DD): ")
                    due_obj = datetime.strptime(due_str, "%Y-%m-%d")
                    self.controller_instance.add_task(task_name, due_obj)
                    print("[UI] Task added.")
                elif user_cmd == 'v':
                    print(self.controller_instance.view_uncompleted_tasks())
                elif user_cmd == 'c':
                    idx = int(input("[UI] Task index to complete: "))
                    self.controller_instance.complete_task(idx)
                    print("[UI] Task completed.")
                elif user_cmd == 'l':
                    file_path = input("[UI] File to load: ")
                    self.controller_instance.load_tasks(file_path)
                    print("[UI] Tasks loaded.")
                elif user_cmd == 's':
```

```
                    file_path = input("[UI] File to save: ")
                    self.controller_instance.save_tasks(file_path)
                    print("[UI] Tasks saved.")
                elif user_cmd == 'q':
                    print("[UI] Exiting application.")
                    break
                else:
                    print("[UI] Invalid menu command.")
            except Exception as e:
                print(f"[UI] Error: {e}")
```

Sample Output:

```
[UI] Main Menu: [A]dd, [V]iew, [C]omplete, [L]oad, [S]ave, [Q]uit
[UI] Command: a
[UI] Task name: Plan trip
[UI] Due date (YYYY-MM-DD): 2025-08-10
[UI] Task added.
[UI] Command: v
The following tasks are still to be done:
Task: Plan trip (Due: 2025-08-10, Completed: False)
```

Explanation:
In this section, the separation of concerns in the to-do application is revealed. Various names of variables and print statements make the role of each class and any of its methods evident. All user interaction is accomplished by CommandLineUI class, whereas controller is responsible of business logic and execution of tasks. TaskList class is used to manage the tasks and DAOs take care of persistence. The code is modular and easily maintainable since each layer involves using well-defined interfaces. This architecture helps to follow the Principle of Single responsibility and causes the application to be extendable. Through this exercise, you get to understand how to structure code in the most logical way that is easy to understand and grow. The techniques used here are important in order to create large maintainable systems. The exercise adds significance to the role of proper delineation of components, and the usefulness of modular design. It is a common practice in the field of professional software development, as well as of teamwork.

Exercise 5: Testing and Debugging the To-Do Application

Introduction:
This exercise covers testing and debugging techniques for the to-do application. You will test all menu options, verify exception handling, and use the debugger to step through key methods.

```
# test_debug.py (snippet)
def test_todo_app():
    from controllers import TaskManagerController
    controller = TaskManagerController()
    print("[Test] Starting To-Do App tests.")
    controller.add_task("Test task A")
```

```
    controller.add_task("Test task B")
    print("[Test] Added two tasks.")
    print(controller.view_uncompleted_tasks())
    try:
        controller.complete_task(999)
    except IndexError as e:
        print(f"[Test] Caught expected error: {e}")
    try:
        controller.load_tasks("nonexistent.csv")
    except FileNotFoundError as e:
        print(f"[Test] Caught expected error: {e}")
    print("[Test] All tests completed.")
```

Sample Output:

```
[Test] Starting To-Do App tests.
[Test] Added two tasks.
The following tasks are still to be done:
Task: Test task A (Due: No due date, Completed: False)
Task: Test task B (Due: No due date, Completed: False)
[Test] Caught expected error: Task index does not exist.
[Test] Caught expected error: File nonexistent.csv not found.
[Test] All tests completed.
```

Explanation:
Here we are interested in testing and debugging of the to-do application. The names given to variables are unique and when there are print statements one can easily trace the flow of execution and find problems easily. The test function contains tasks that it adds, observes and intentionally causes errors to check exception handling. The code has also passed the test of sturdy error handling by trapping and reporting anticipated errors. Program flow The debugger also can be used to step through methods to improve your understanding of program flow and to expose bugs. This practice goes to demonstrate the relevance of rigorous testing and debugging in development of software. The competencies acquired here are critical towards development of dependable, sustainable applications. Another thing that can be seen in the task is the importance of descriptive and understandable output and user feedbacks. With these techniques, you will be a better and competent programmer. Testing and debugging play an important role in the delivery of quality software within any work environment.

Conclusion:

The exercise in this week has also consolidated the best practices of software engineering using SOLID principles, exception management, data abstraction, and testing. You constructed a to-do application that had good error handling, modularity and explicit separation of concern. All of the exercises were used to cement the lesson on unique identifiers, descriptive output and maintainable code. These skills are important in creation of scalable reliable and easy to use software. Continue and perfect the method to be able to be a confident and competent developer!