Exercise 2: Problem Statements for Mini-projects

1. MUST Comply: No fancy looking application is required to be built as part of this exercise. It shall be a simple console/terminal based application. Focus shall ONLY be on logic and code quality as described in the points below.
2. MUST Comply: Coding should be done adopting best practices - Behavioural/structural/creational design patterns, SOLID design principles, OOPs programming, language of candidates choice.
3. Candidate shall pick ONE among the EIGHT problem statements provided below, and solve.
4. Note: Please feel free to assume unknowns, and be creative in enhancing the problem statements to demonstrate your excellence!
5. Astronaut Daily Schedule Organizer Programming Exercise Problem Statement Design and implement a console-based application that helps astronauts organize their daily schedules. The application should allow users to add, remove, and view daily tasks. Each task will have a description, start time, end time, and priority level. The intent behind this problem statement is to evaluate your ability to implement a basic CRUD (Create, Read, Update, Delete) application, manage data efficiently, and apply best coding practices. Functional Requirements Mandatory Requirements 1. Add a new task with description, start time, end time, and priority level.

2. Remove an existing task. 3. View all tasks sorted by start time. 4. Validate that new tasks do not overlap with existing tasks. 5. Provide appropriate error messages for invalid operations. Optional Requirements 1. Edit an existing task. 2. Mark tasks as completed. 3. View tasks for a specific priority level. Non-functional Requirements 1. The application should handle exceptions gracefully. 2. Ensure the application is optimized for performance. 3. Implement a logging mechanism for tracking application usage and errors. Key Focus Design Patterns to be used 1. Singleton Pattern: Ensure there is only one instance of the schedule manager. 2. Factory Pattern: Use a factory to create task objects. 3. Observer Pattern: Notify users of task conflicts or updates. Detailed Instructions 1. Use the Singleton Pattern to create a ScheduleManager class that manages all tasks. 2. Implement a TaskFactory class to create Task objects. 3. Use the Observer Pattern to alert users if a new task conflicts with an existing one. Possible Inputs and Corresponding Outputs Positive Cases 1. Input: Add Task("Morning Exercise", "07:00", "08:00", "High") Output: Task added successfully. No conflicts. 2. Input: Add Task("Team Meeting", "09:00", "10:00", "Medium") Output: Task added successfully. No conflicts. 3. Input: View Tasks Output: a. 07:00 - 08:00: Morning Exercise [High] b. 09:00 - 10:00: Team Meeting [Medium] 4. Input: Remove Task("Morning Exercise") Output: Task removed successfully. 5. Input: Add Task("Lunch Break", "12:00", "13:00", "Low") Output: Task added successfully. No conflicts. Negative Cases 1. Input: Add Task("Training Session", "09:30", "10:30", "High") Output: Error: Task conflicts with existing task "Team Meeting". 2. Input: Remove Task("Non-existent Task") Output: Error: Task not found. 3. Input: Add Task("Invalid Time Task", "25:00", "26:00", "Low") Output: Error: Invalid time format. 4. Input: Add Task("Overlap Task", "08:30", "09:30", "Medium") Output: Error: Task conflicts with existing task "Team Meeting". 5. Input: View Tasks (when no tasks exist) Output: No tasks scheduled for the day. Evaluation 1. Code Quality: Adherence to best practices, use of design patterns, SOLID principles, and OOP. 2. Functionality: All mandatory requirements implemented correctly. 3. Error Handling: Graceful handling of all errors and edge cases. 4. Performance: Code is optimized for performance. 5. Explanation: Candidate's ability to walk through the code and explain design decisions and logic. 6. Documentation: Code is well-documented, and usage instructions are clear. The goal of this exercise is to assess the candidate's coding skills, understanding of design patterns, and ability to produce high-quality, maintainable code.

from datetime import datetime, timedelta

import logging

# Setting up logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

# Task class

class Task:

def \_\_init\_\_(self, description, start\_time, end\_time, priority):

self.description = description

self.start\_time = start\_time

self.end\_time = end\_time

self.priority = priority

self.completed = False

def \_\_str\_\_(self):

status = "Completed" if self.completed else "Pending"

return f"{self.start\_time.strftime('%H:%M')} - {self.end\_time.strftime('%H:%M')}: {self.description} [{self.priority}] - {status}"

# TaskFactory class

class TaskFactory:

@staticmethod

def create\_task(description, start\_time, end\_time, priority):

try:

start\_time = datetime.strptime(start\_time, '%H:%M')

end\_time = datetime.strptime(end\_time, '%H:%M')

if start\_time >= end\_time:

raise ValueError("Start time must be before end time")

return Task(description, start\_time, end\_time, priority)

except ValueError as e:

logging.error(f"Error creating task: {e}")

return None

# Singleton ScheduleManager class

class ScheduleManager:

\_instance = None

@staticmethod

def get\_instance():

if ScheduleManager.\_instance is None:

ScheduleManager()

return ScheduleManager.\_instance

def \_\_init\_\_(self):

if ScheduleManager.\_instance is not None:

raise Exception("This class is a singleton!")

else:

ScheduleManager.\_instance = self

self.tasks = []

self.observers = []

def add\_task(self, task):

if self.\_validate\_task(task):

self.tasks.append(task)

self.tasks.sort(key=lambda x: x.start\_time)

logging.info(f"Task '{task.description}' added successfully. No conflicts.")

self.\_notify\_observers()

else:

logging.error(f"Error: Task '{task.description}' conflicts with existing tasks.")

def remove\_task(self, description):

task\_to\_remove = next((task for task in self.tasks if task.description == description), None)

if task\_to\_remove:

self.tasks.remove(task\_to\_remove)

logging.info(f"Task '{description}' removed successfully.")

self.\_notify\_observers()

else:

logging.error(f"Error: Task '{description}' not found.")

def view\_tasks(self):

if not self.tasks:

logging.info("No tasks scheduled for the day.")

for task in self.tasks:

print(task)

def edit\_task(self, old\_description, new\_task):

self.remove\_task(old\_description)

self.add\_task(new\_task)

def mark\_task\_completed(self, description):

task = next((task for task in self.tasks if task.description == description), None)

if task:

task.completed = True

logging.info(f"Task '{description}' marked as completed.")

else:

logging.error(f"Error: Task '{description}' not found.")

def view\_tasks\_by\_priority(self, priority):

tasks\_by\_priority = [task for task in self.tasks if task.priority == priority]

if not tasks\_by\_priority:

logging.info(f"No tasks with priority '{priority}' found.")

for task in tasks\_by\_priority:

print(task)

def \_validate\_task(self, new\_task):

for task in self.tasks:

if not (new\_task.end\_time <= task.start\_time or new\_task.start\_time >= task.end\_time):

return False

return True

def add\_observer(self, observer):

self.observers.append(observer)

def remove\_observer(self, observer):

self.observers.remove(observer)

def \_notify\_observers(self):

for observer in self.observers:

observer.update(self.tasks)

# ConflictObserver class

class ConflictObserver:

def update(self, tasks):

conflicts = self.\_find\_conflicts(tasks)

if conflicts:

logging.warning("Task conflicts detected:")

for conflict in conflicts:

logging.warning(conflict)

def \_find\_conflicts(self, tasks):

conflicts = []

for i in range(len(tasks) - 1):

if tasks[i].end\_time > tasks[i + 1].start\_time:

conflicts.append(f"{tasks[i]} conflicts with {tasks[i + 1]}")

return conflicts

# Main Application

def main():

schedule\_manager = ScheduleManager.get\_instance()

conflict\_observer = ConflictObserver()

schedule\_manager.add\_observer(conflict\_observer)

while True:

print("\nAstronaut Daily Schedule Organizer")

print("1. Add Task")

print("2. Remove Task")

print("3. View All Tasks")

print("4. Edit Task")

print("5. Mark Task as Completed")

print("6. View Tasks by Priority")

print("7. Exit")

choice = input("Enter your choice: ")

if choice == '1':

description = input("Enter task description: ")

start\_time = input("Enter start time (HH:MM): ")

end\_time = input("Enter end time (HH:MM): ")

priority = input("Enter priority (High/Medium/Low): ")

task = TaskFactory.create\_task(description, start\_time, end\_time, priority)

if task:

schedule\_manager.add\_task(task)

elif choice == '2':

description = input("Enter task description to remove: ")

schedule\_manager.remove\_task(description)

elif choice == '3':

schedule\_manager.view\_tasks()

elif choice == '4':

old\_description = input("Enter the description of the task to edit: ")

description = input("Enter new task description: ")

start\_time = input("Enter new start time (HH:MM): ")

end\_time = input("Enter new end time (HH:MM): ")

priority = input("Enter new priority (High/Medium/Low): ")

task = TaskFactory.create\_task(description, start\_time, end\_time, priority)

if task:

schedule\_manager.edit\_task(old\_description, task)

elif choice == '5':

description = input("Enter the description of the task to mark as completed: ")

schedule\_manager.mark\_task\_completed(description)

elif choice == '6':

priority = input("Enter priority (High/Medium/Low) to view tasks: ")

schedule\_manager.view\_tasks\_by\_priority(priority)

elif choice == '7':

break

else:

logging.error("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

main()