1. Astronaut Daily Schedule Organizer Programming Exercise

#include <iostream>

#include <string>

#include <vector>

#include <algorithm>

#include <map>

#include <sstream>

#include <iomanip>

#include <memory>

#include <set>

bool isValidTime(const std::string& time) {

if (time.length() != 5 || time[2] != ':') return false;

int h = std::stoi(time.substr(0, 2));

int m = std::stoi(time.substr(3, 2));

return h >= 0 && h <= 23 && m >= 0 && m <= 59;

}

bool timeLess(const std::string& t1, const std::string& t2) {

return t1 < t2;

}

bool timeOverlap(const std::string& start1, const std::string& end1, const std::string& start2, const std::string& end2) {

return !(end1 <= start2 || start1 >= end2);

}

// Task Class

class Task {

private:

std::string description;

std::string startTime;

std::string endTime;

std::string priority;

bool completed;

public:

Task(const std::string& desc, const std::string& start, const std::string& end, const std::string& prio)

: description(desc), startTime(start), endTime(end), priority(prio), completed(false) {}

std::string getDescription() const { return description; }

std::string getStartTime() const { return startTime; }

std::string getEndTime() const { return endTime; }

std::string getPriority() const { return priority; }

bool isCompleted() const { return completed; }

void markCompleted() { completed = true; }

void printTask() const {

std::cout << startTime << " - " << endTime << ": " << description << " [" << priority << "]";

if (completed) std::cout << " [Completed]";

std::cout << std::endl;

}

};

//Task Factory (Factory Pattern)

class TaskFactory {

public:

static std::shared\_ptr<Task> createTask(const std::string& desc, const std::string& start, const std::string& end, const std::string& priority) {

if (!isValidTime(start) || !isValidTime(end) || start >= end) {

throw std::invalid\_argument("Error: Invalid time format or time range.");

}

return std::make\_shared<Task>(desc, start, end, priority);

}

};

//Observer(Observer Pattern)

class Observer {

public:

virtual void notifyConflict(const std::string& conflictTaskDesc) = 0;

};

class ConflictNotifier : public Observer {

public:

void notifyConflict(const std::string& conflictTaskDesc) override {

std::cout << "Error: Task conflicts with existing task \"" << conflictTaskDesc << "\".\n";

}

};

// ScheduleManager (Singleton Pattern)

class ScheduleManager {

private:

static ScheduleManager\* instance;

std::vector<std::shared\_ptr<Task>> tasks;

std::vector<Observer\*> observers;

ScheduleManager() {}

public:

static ScheduleManager\* getInstance() {

if (!instance) {

instance = new ScheduleManager();

}

return instance;

}

void attachObserver(Observer\* obs) {

observers.push\_back(obs);

}

void notifyObservers(const std::string& conflictTask) {

for (auto obs : observers) {

obs->notifyConflict(conflictTask);

}

}

bool addTask(std::shared\_ptr<Task> newTask) {

for (auto& task : tasks) {

if (timeOverlap(task->getStartTime(), task->getEndTime(), newTask>getStartTime(), newTask->getEndTime())) {

notifyObservers(task->getDescription());

return false;

}

}

tasks.push\_back(newTask);

std::cout << "Task added successfully. No conflicts.\n";

return true;

}

void removeTask(const std::string& description) {

auto it = std::remove\_if(tasks.begin(), tasks.end(),

[&description](const std::shared\_ptr<Task>& task) {

return task->getDescription() == description;

});

if (it == tasks.end()) {

std::cout << "Error: Task not found.\n";

} else {

tasks.erase(it, tasks.end());

std::cout << "Task removed successfully.\n";

}

}

void viewTasks() {

if (tasks.empty()) {

std::cout << "No tasks scheduled for the day.\n";

return;

}

std::sort(tasks.begin(), tasks.end(),

[](const std::shared\_ptr<Task>& a, const std::shared\_ptr<Task>& b) {

return timeLess(a->getStartTime(), b->getStartTime());

});

for (const auto& task : tasks) {

task->printTask();

}

}

void viewByPriority(const std::string& priority) {

bool found = false;

for (const auto& task : tasks) {

if (task->getPriority() == priority) {

task->printTask();

found = true;

}

}

if (!found) std::cout << "No tasks found with priority: " << priority << "\n";

}

void markTaskCompleted(const std::string& description) {

for (auto& task : tasks) {

if (task->getDescription() == description) {

task->markCompleted();

std::cout << "Task marked as completed.\n";

return;

}

}

std::cout << "Error: Task not found.\n";

}

};

ScheduleManager\* ScheduleManager::instance = nullptr;

// Application UI

void showMenu() {

std::cout << "\n=== Astronaut Daily Schedule Organizer ===\n";

std::cout << "1. Add Task\n";

std::cout << "2. Remove Task\n";

std::cout << "3. View Tasks\n";

std::cout << "4. View Tasks by Priority\n";

std::cout << "5. Mark Task as Completed\n";

std::cout << "6. Exit\n";

std::cout << "Choose an option: ";

}

int main() {

auto\* manager = ScheduleManager::getInstance();

ConflictNotifier notifier;

manager->attachObserver(&notifier);

int choice;

while (true) {

showMenu();

std::cin >> choice;

std::cin.ignore(); // clear buffer

try {

if (choice == 1) {

std::string desc, start, end, priority;

std::cout << "Enter description: ";

std::getline(std::cin, desc);

std::cout << "Enter start time (HH:MM): ";

std::getline(std::cin, start);

std::cout << "Enter end time (HH:MM): ";

std::getline(std::cin, end);

std::cout << "Enter priority (High/Medium/Low): ";

std::getline(std::cin, priority);

auto task = TaskFactory::createTask(desc, start, end, priority);

manager->addTask(task);

}

else if (choice == 2) {

std::string desc;

std::cout << "Enter task description to remove: ";

std::getline(std::cin, desc);

manager->removeTask(desc);

}

else if (choice == 3) {

manager->viewTasks();

}

else if (choice == 4) {

std::string prio;

std::cout << "Enter priority level (High/Medium/Low): ";

std::getline(std::cin, prio);

manager->viewByPriority(prio);

}

else if (choice == 5) {

std::string desc;

std::cout << "Enter task description to mark as completed: ";

std::getline(std::cin, desc);

manager->markTaskCompleted(desc);

}

else if (choice == 6) {

std::cout << "Exiting...\n";

break;

}

else {

std::cout << "Invalid choice.\n";

}

}

catch (const std::exception& ex) {

std::cout << ex.what() << "\n";

}

}

return 0;

}