using System;

public class Task

{

public string Description { get; set; }

public TimeSpan StartTime { get; set; }

public TimeSpan EndTime { get; set; }

public string Priority { get; set; }

public Task(string description, TimeSpan startTime, TimeSpan endTime, string priority)

{

Description = description;

StartTime = startTime;

EndTime = endTime;

Priority = priority;

}

public override string ToString()

{

return $"{StartTime} - {EndTime}: {Description} [{Priority}]";

}

}

public static class TaskFactory

{

public static Task CreateTask(string description, TimeSpan startTime, TimeSpan endTime, string priority)

{

return new Task(description, startTime, endTime, priority);

}

}

using System;

using System.Collections.Generic;

using System.Linq;

public class ScheduleManager

{

private static ScheduleManager \_instance;

private static readonly object \_lock = new object();

private List<Task> \_tasks = new List<Task>();

private ScheduleManager() { }

public static ScheduleManager Instance

{

get

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new ScheduleManager();

}

return \_instance;

}

}

}

public bool AddTask(Task task, out string message)

{

if (\_tasks.Any(t => t.StartTime < task.EndTime && task.StartTime < t.EndTime))

{

message = $"Error: Task conflicts with existing task '{\_tasks.First(t => t.StartTime < task.EndTime && task.StartTime < t.EndTime).Description}'";

return false;

}

\_tasks.Add(task);

\_tasks = \_tasks.OrderBy(t => t.StartTime).ToList(); // Sorting by start time

message = "Task added successfully. No conflicts.";

return true;

}

public bool RemoveTask(string description, out string message)

{

var taskToRemove = \_tasks.FirstOrDefault(t => t.Description == description);

if (taskToRemove != null)

{

\_tasks.Remove(taskToRemove);

message = "Task removed successfully.";

return true;

}

message = "Error: Task not found.";

return false;

}

public List<Task> GetTasks()

{

return \_tasks;

}

public List<Task> GetTasksByPriority(string priority)

{

return \_tasks.Where(t => t.Priority.Equals(priority, StringComparison.OrdinalIgnoreCase)).ToList();

}

}

using System;

class Program

{

static void Main()

{

ScheduleManager manager = ScheduleManager.Instance;

while (true)

{

Console.WriteLine("1. Add Task");

Console.WriteLine("2. Remove Task");

Console.WriteLine("3. View Tasks");

Console.WriteLine("4. View Tasks by Priority");

Console.WriteLine("5. Exit");

Console.Write("Select an option: ");

string option = Console.ReadLine();

switch (option)

{

case "1":

AddTask(manager);

break;

case "2":

RemoveTask(manager);

break;

case "3":

ViewTasks(manager);

break;

case "4":

ViewTasksByPriority(manager);

break;

case "5":

return;

default:

Console.WriteLine("Invalid option. Please try again.");

break;

}

}

}

static void AddTask(ScheduleManager manager)

{

try

{

Console.Write("Enter task description: ");

string description = Console.ReadLine();

Console.Write("Enter start time (HH:MM): ");

TimeSpan startTime = TimeSpan.Parse(Console.ReadLine());

Console.Write("Enter end time (HH:MM): ");

TimeSpan endTime = TimeSpan.Parse(Console.ReadLine());

Console.Write("Enter priority (Low/Medium/High): ");

string priority = Console.ReadLine();

Task task = TaskFactory.CreateTask(description, startTime, endTime, priority);

string message;

if (manager.AddTask(task, out message))

{

Console.WriteLine(message);

}

else

{

Console.WriteLine(message);

}

}

catch (FormatException)

{

Console.WriteLine("Error: Invalid time format.");

}

}

static void RemoveTask(ScheduleManager manager)

{

Console.Write("Enter task description to remove: ");

string description = Console.ReadLine();

string message;

if (manager.RemoveTask(description, out message))

{

Console.WriteLine(message);

}

else

{

Console.WriteLine(message);

}

}

static void ViewTasks(ScheduleManager manager)

{

var tasks = manager.GetTasks();

if (tasks.Count == 0)

{

Console.WriteLine("No tasks scheduled for the day.");

}

else

{

foreach (var task in tasks)

{

Console.WriteLine(task);

}

}

}

static void ViewTasksByPriority(ScheduleManager manager)

{

Console.Write("Enter priority level (Low/Medium/High): ");

string priority = Console.ReadLine();

var tasks = manager.GetTasksByPriority(priority);

if (tasks.Count == 0)

{

Console.WriteLine($"No tasks found with priority '{priority}'.");

}

else

{

foreach (var task in tasks)

{

Console.WriteLine(task);

}

}

}

}

2.

using System;

using System.Collections.Generic;

using System.Linq;

public class OfficeManager

{

private static OfficeManager \_instance;

private static readonly object \_lock = new object();

private Dictionary<int, ConferenceRoom> \_rooms = new Dictionary<int, ConferenceRoom>();

private OfficeManager() { }

public static OfficeManager Instance

{

get

{

lock (\_lock)

{

if (\_instance == null)

{

\_instance = new OfficeManager();

}

return \_instance;

}

}

}

public void ConfigureRooms(int numberOfRooms)

{

\_rooms.Clear();

for (int i = 1; i <= numberOfRooms; i++)

{

\_rooms[i] = new ConferenceRoom(i);

}

}

public ConferenceRoom GetRoom(int roomId)

{

return \_rooms.ContainsKey(roomId) ? \_rooms[roomId] : null;

}

public IEnumerable<ConferenceRoom> GetAllRooms()

{

return \_rooms.Values;

}

}

using System;

using System.Collections.Generic;

public interface IObserver

{

void Update(ConferenceRoom room);

}

public interface IObservable

{

void AddObserver(IObserver observer);

void RemoveObserver(IObserver observer);

void NotifyObservers();

}

public class ConferenceRoom : IObservable

{

public int Id { get; }

public string BookingStatus { get; private set; }

public bool IsOccupied { get; private set; }

private List<IObserver> \_observers = new List<IObserver>();

private DateTime? \_lastOccupiedTime;

public ConferenceRoom(int id)

{

Id = id;

BookingStatus = "Available";

}

public void BookRoom()

{

BookingStatus = "Booked";

\_lastOccupiedTime = null;

}

public void CancelBooking()

{

BookingStatus = "Available";

\_lastOccupiedTime = null;

UpdateOccupancyStatus(false);

}

public void UpdateOccupancyStatus(bool occupied)

{

IsOccupied = occupied;

\_lastOccupiedTime = occupied ? (DateTime?)DateTime.Now : null;

NotifyObservers();

}

public void AddObserver(IObserver observer)

{

\_observers.Add(observer);

}

public void RemoveObserver(IObserver observer)

{

\_observers.Remove(observer);

}

public void NotifyObservers()

{

foreach (var observer in \_observers)

{

observer.Update(this);

}

}

public void CheckAutoRelease()

{

if (!IsOccupied && \_lastOccupiedTime.HasValue && DateTime.Now.Subtract(\_lastOccupiedTime.Value).TotalMinutes >= 5)

{

CancelBooking();

}

}

}

using System;

public interface ICommand

{

void Execute();

}

public class BookRoomCommand : ICommand

{

private readonly ConferenceRoom \_room;

public BookRoomCommand(ConferenceRoom room)

{

\_room = room;

}

public void Execute()

{

if (\_room.BookingStatus == "Available")

{

\_room.BookRoom();

Console.WriteLine($"Room {\_room.Id} booked successfully.");

}

else

{

Console.WriteLine($"Room {\_room.Id} is already booked.");

}

}

}

public class CancelBookingCommand : ICommand

{

private readonly ConferenceRoom \_room;

public CancelBookingCommand(ConferenceRoom room)

{

\_room = room;

}

public void Execute()

{

if (\_room.BookingStatus == "Booked")

{

\_room.CancelBooking();

Console.WriteLine($"Booking for room {\_room.Id} canceled.");

}

else

{

Console.WriteLine($"Room {\_room.Id} is not booked.");

}

}

}

using System;

public class LightControl : IObserver

{

public void Update(ConferenceRoom room)

{

if (room.IsOccupied)

{

Console.WriteLine($"Lights in room {room.Id} are turned ON.");

}

else

{

Console.WriteLine($"Lights in room {room.Id} are turned OFF.");

}

}

}

public class ACControl : IObserver

{

public void Update(ConferenceRoom room)

{

if (room.IsOccupied)

{

Console.WriteLine($"AC in room {room.Id} is turned ON.");

}

else

{

Console.WriteLine($"AC in room {room.Id} is turned OFF.");

}

}

}

using System;

class Program

{

static void Main()

{

var officeManager = OfficeManager.Instance;

officeManager.ConfigureRooms(3);

// Setup room controls

foreach (var room in officeManager.GetAllRooms())

{

room.AddObserver(new LightControl());

room.AddObserver(new ACControl());

}

while (true)

{

Console.WriteLine("1. Book Room");

Console.WriteLine("2. Cancel Booking");

Console.WriteLine("3. Check Auto Release");

Console.WriteLine("4. Exit");

Console.Write("Select an option: ");

string option = Console.ReadLine();

switch (option)

{

case "1":

BookRoom(officeManager);

break;

case "2":

CancelBooking(officeManager);

break;

case "3":

CheckAutoRelease(officeManager);

break;

case "4":

return;

default:

Console.WriteLine("Invalid option. Please try again.");

break;

}

}

}

static void BookRoom(OfficeManager officeManager)

{

Console.Write("Enter room ID to book: ");

if (int.TryParse(Console.ReadLine(), out int roomId))

{

var room = officeManager.GetRoom(roomId);

if (room != null)

{

var command = new BookRoomCommand(room);

command.Execute();

}

else

{

Console.WriteLine("Room not found.");

}

}

else

{

Console.WriteLine("Invalid room ID.");

}

}

static void CancelBooking(OfficeManager officeManager)

{

Console.Write("Enter room ID to cancel booking: ");

if (int.TryParse(Console.ReadLine(), out int roomId))

{

var room = officeManager.GetRoom(roomId);

if (room != null)

{

var command = new CancelBookingCommand(room);

command.Execute();

}

else

{

Console.WriteLine("Room not found.");

}

}

else

{

Console.WriteLine("Invalid room ID.");

}

}

static void CheckAutoRelease(OfficeManager officeManager)

{

foreach (var room in officeManager.GetAllRooms())

{

room.CheckAutoRelease();

}

}

}

3. public enum Direction

{

North,

East,

South,

West

}

using System.Collections.Generic;

public class Grid

{

public int Width { get; }

public int Height { get; }

private HashSet<(int x, int y)> \_obstacles = new HashSet<(int x, int y)>();

public Grid(int width, int height)

{

Width = width;

Height = height;

}

public void AddObstacle(int x, int y)

{

\_obstacles.Add((x, y));

}

public bool IsObstacle(int x, int y)

{

return \_obstacles.Contains((x, y));

}

public bool IsWithinBounds(int x, int y)

{

return x >= 0 && x < Width && y >= 0 && y < Height;

}

}

using System;

public class Rover

{

private int \_x;

private int \_y;

private Direction \_direction;

private readonly Grid \_grid;

public Rover(int x, int y, Direction direction, Grid grid)

{

\_x = x;

\_y = y;

\_direction = direction;

\_grid = grid;

}

public void Move()

{

int newX = \_x;

int newY = \_y;

switch (\_direction)

{

case Direction.North:

newY += 1;

break;

case Direction.East:

newX += 1;

break;

case Direction.South:

newY -= 1;

break;

case Direction.West:

newX -= 1;

break;

}

if (\_grid.IsWithinBounds(newX, newY) && !\_grid.IsObstacle(newX, newY))

{

\_x = newX;

\_y = newY;

}

}

public void TurnLeft()

{

\_direction = \_direction switch

{

Direction.North => Direction.West,

Direction.East => Direction.North,

Direction.South => Direction.East,

Direction.West => Direction.South,

\_ => \_direction

};

}

public void TurnRight()

{

\_direction = \_direction switch

{

Direction.North => Direction.East,

Direction.East => Direction.South,

Direction.South => Direction.West,

Direction.West => Direction.North,

\_ => \_direction

};

}

public (int x, int y, Direction direction) GetStatus()

{

return (\_x, \_y, \_direction);

}

}

public interface ICommand

{

void Execute();

}

public class MoveCommand : ICommand

{

private readonly Rover \_rover;

public MoveCommand(Rover rover)

{

\_rover = rover;

}

public void Execute()

{

\_rover.Move();

}

}

public class TurnLeftCommand : ICommand

{

private readonly Rover \_rover;

public TurnLeftCommand(Rover rover)

{

\_rover = rover;

}

public void Execute()

{

\_rover.TurnLeft();

}

}

public class TurnRightCommand : ICommand

{

private readonly Rover \_rover;

public TurnRightCommand(Rover rover)

{

\_rover = rover;

}

public void Execute()

{

\_rover.TurnRight();

}

}

using System.Collections.Generic;

public class CommandExecutor

{

private readonly List<ICommand> \_commands = new List<ICommand>();

public void AddCommand(ICommand command)

{

\_commands.Add(command);

}

public void ExecuteCommands()

{

foreach (var command in \_commands)

{

command.Execute();

}

}

}

using System;

class Program

{

static void Main()

{

var grid = new Grid(10, 10);

grid.AddObstacle(2, 2);

grid.AddObstacle(3, 5);

var rover = new Rover(0, 0, Direction.North, grid);

var commandExecutor = new CommandExecutor();

commandExecutor.AddCommand(new MoveCommand(rover)); // M

commandExecutor.AddCommand(new MoveCommand(rover)); // M

commandExecutor.AddCommand(new TurnRightCommand(rover)); // R

commandExecutor.AddCommand(new MoveCommand(rover)); // M

commandExecutor.AddCommand(new TurnLeftCommand(rover)); // L

commandExecutor.AddCommand(new MoveCommand(rover)); // M

commandExecutor.ExecuteCommands();

var status = rover.GetStatus();

Console.WriteLine($"Final Position: ({status.x}, {status.y}, {status.direction})");

// Optional Status Report

Console.WriteLine($"Status Report: Rover is at ({status.x}, {status.y}) facing {status.direction}. No Obstacles detected.");

}

}

4. from abc import ABC, abstractmethod

from datetime import datetime

import time

from typing import List, Dict, Any

# Observer Pattern

class Observer(ABC):

@abstractmethod

def update(self, message: str):

pass

class Subject(ABC):

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

self.\_observers = []

def add\_observer(self, observer: Observer):

self.\_observers.append(observer)

def remove\_observer(self, observer: Observer):

self.\_observers.remove(observer)

def notify\_observers(self, message: str):

for observer in self.\_observers:

observer.update(message)

# Factory Method

class DeviceFactory(ABC):

@abstractmethod

def create\_device(self, device\_id: int, \*\*kwargs) -> 'Device':

pass

class ConcreteDeviceFactory(DeviceFactory):

def create\_device(self, device\_id: int, \*\*kwargs) -> 'Device':

device\_type = kwargs.get('type')

if device\_type == 'light':

return Light(device\_id, \*\*kwargs)

elif device\_type == 'thermostat':

return Thermostat(device\_id, \*\*kwargs)

elif device\_type == 'door':

return DoorLock(device\_id, \*\*kwargs)

else:

raise ValueError(f"Unknown device type: {device\_type}")

# Proxy Pattern

class DeviceProxy(ABC):

@abstractmethod

def turn\_on(self):

pass

@abstractmethod

def turn\_off(self):

pass

@abstractmethod

def status(self) -> str:

pass

class Device(ABC):

def \_\_init\_\_(self, device\_id: int):

self.device\_id = device\_id

@abstractmethod

def turn\_on(self):

pass

@abstractmethod

def turn\_off(self):

pass

@abstractmethod

def status(self) -> str:

pass

class Light(Device, DeviceProxy):

def \_\_init\_\_(self, device\_id: int, status: str = 'off'):

super().\_\_init\_\_(device\_id)

self.\_status = status

def turn\_on(self):

self.\_status = 'on'

return f"Light {self.device\_id} is On."

def turn\_off(self):

self.\_status = 'off'

return f"Light {self.device\_id} is Off."

def status(self) -> str:

return f"Light {self.device\_id} is {self.\_status}."

class Thermostat(Device, DeviceProxy):

def \_\_init\_\_(self, device\_id: int, temperature: int = 70):

super().\_\_init\_\_(device\_id)

self.\_temperature = temperature

def turn\_on(self):

# Thermostats don't really turn on or off, but we could adjust temperature here.

return f"Thermostat {self.device\_id} is set to {self.\_temperature} degrees."

def turn\_off(self):

return f"Thermostat {self.device\_id} cannot be turned off."

def status(self) -> str:

return f"Thermostat {self.device\_id} is set to {self.\_temperature} degrees."

class DoorLock(Device, DeviceProxy):

def \_\_init\_\_(self, device\_id: int, status: str = 'locked'):

super().\_\_init\_\_(device\_id)

self.\_status = status

def turn\_on(self):

self.\_status = 'locked'

return f"Door {self.device\_id} is Locked."

def turn\_off(self):

self.\_status = 'unlocked'

return f"Door {self.device\_id} is Unlocked."

def status(self) -> str:

return f"Door {self.device\_id} is {self.\_status}."

# Smart Home System

class SmartHomeSystem(Observer):

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

self.devices: Dict[int, Device] = {}

self.scheduled\_tasks: List[Dict[str, Any]] = []

self.automated\_triggers: List[Dict[str, Any]] = []

self.current\_time = datetime.now()

def add\_device(self, device: Device):

self.devices[device.device\_id] = device

def remove\_device(self, device\_id: int):

if device\_id in self.devices:

del self.devices[device\_id]

def turn\_on\_device(self, device\_id: int):

if device\_id in self.devices:

print(self.devices[device\_id].turn\_on())

self.notify\_observers(f"Device {device\_id} turned on")

def turn\_off\_device(self, device\_id: int):

if device\_id in self.devices:

print(self.devices[device\_id].turn\_off())

self.notify\_observers(f"Device {device\_id} turned off")

def set\_schedule(self, device\_id: int, time: str, command: str):

self.scheduled\_tasks.append({'device': device\_id, 'time': time, 'command': command})

print(f"Scheduled {command} for device {device\_id} at {time}")

def add\_trigger(self, condition: str, action: str):

self.automated\_triggers.append({'condition': condition, 'action': action})

print(f"Added trigger: If {condition}, then {action}")

def check\_schedules(self):

current\_time\_str = self.current\_time.strftime("%H:%M")

for task in self.scheduled\_tasks:

if task['time'] == current\_time\_str:

if task['command'] == 'Turn On':

self.turn\_on\_device(task['device'])

elif task['command'] == 'Turn Off':

self.turn\_off\_device(task['device'])

def check\_triggers(self):

for trigger in self.automated\_triggers:

# This example only handles a basic temperature condition.

if trigger['condition'] == 'temperature > 75':

if isinstance(self.devices[2], Thermostat) and self.devices[2].\_temperature > 75:

eval(trigger['action']) # Execute the action

def update(self, message: str):

print(f"System Update: {message}")

def status\_report(self):

report = [device.status() for device in self.devices.values()]

print(" | ".join(report))

def run(self):

while True:

self.check\_schedules()

self.check\_triggers()

self.current\_time = datetime.now()

time.sleep(60) # Update every minute

# Example usage

factory = ConcreteDeviceFactory()

smart\_home = SmartHomeSystem()

# Initialize devices

light1 = factory.create\_device(1, type='light', status='off')

thermostat = factory.create\_device(2, type='thermostat', temperature=70)

door = factory.create\_device(3, type='door', status='locked')

# Add devices to the system

smart\_home.add\_device(light1)

smart\_home.add\_device(thermostat)

smart\_home.add\_device(door)

# Add schedules and triggers

smart\_home.set\_schedule(2, "06:00", "Turn On")

smart\_home.add\_trigger("temperature > 75", "smart\_home.turn\_off\_device(1)")

# Print status report

smart\_home.status\_report()

# Run the system (commented out to avoid an infinite loop in this environment)

# smart\_home.run()

5. using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading;

namespace RealTimeChatApplication

{

// Observer Pattern

public interface IObserver

{

void Update(string message);

}

public interface ISubject

{

void AddObserver(IObserver observer);

void RemoveObserver(IObserver observer);

void NotifyObservers(string message);

}

public class ChatRoom : ISubject

{

private readonly List<IObserver> \_observers = new List<IObserver>();

private readonly List<string> \_messages = new List<string>();

private readonly Dictionary<string, User> \_users = new Dictionary<string, User>();

public string RoomId { get; }

public ChatRoom(string roomId)

{

RoomId = roomId;

}

public void AddUser(User user)

{

if (!\_users.ContainsKey(user.Username))

{

\_users.Add(user.Username, user);

user.SetChatRoom(this);

NotifyObservers($"{user.Username} has joined the room.");

}

}

public void RemoveUser(User user)

{

if (\_users.ContainsKey(user.Username))

{

\_users.Remove(user.Username);

NotifyObservers($"{user.Username} has left the room.");

}

}

public void SendMessage(User sender, string message)

{

var formattedMessage = $"{sender.Username}: {message}";

\_messages.Add(formattedMessage);

NotifyObservers(formattedMessage);

}

public IEnumerable<string> GetActiveUsers()

{

return \_users.Keys.ToList();

}

public IEnumerable<string> GetMessages()

{

return \_messages.ToList();

}

public void AddObserver(IObserver observer)

{

\_observers.Add(observer);

}

public void RemoveObserver(IObserver observer)

{

\_observers.Remove(observer);

}

public void NotifyObservers(string message)

{

foreach (var observer in \_observers)

{

observer.Update(message);

}

}

}

public class User : IObserver

{

public string Username { get; }

private ChatRoom \_chatRoom;

public User(string username)

{

Username = username;

}

public void SetChatRoom(ChatRoom chatRoom)

{

\_chatRoom = chatRoom;

\_chatRoom.AddObserver(this);

}

public void JoinChatRoom(ChatRoom chatRoom)

{

chatRoom.AddUser(this);

}

public void LeaveChatRoom()

{

\_chatRoom?.RemoveUser(this);

\_chatRoom?.RemoveObserver(this);

\_chatRoom = null;

}

public void SendMessage(string message)

{

\_chatRoom?.SendMessage(this, message);

}

public void Update(string message)

{

Console.WriteLine($"User {Username} received: {message}");

}

}

public class ChatRoomManager

{

private static ChatRoomManager \_instance;

private readonly Dictionary<string, ChatRoom> \_chatRooms = new Dictionary<string, ChatRoom>();

private ChatRoomManager() { }

public static ChatRoomManager Instance => \_instance ??= new ChatRoomManager();

public ChatRoom CreateChatRoom(string roomId)

{

if (!\_chatRooms.ContainsKey(roomId))

{

var chatRoom = new ChatRoom(roomId);

\_chatRooms[roomId] = chatRoom;

return chatRoom;

}

throw new Exception("Chat room already exists.");

}

public ChatRoom GetChatRoom(string roomId)

{

\_chatRooms.TryGetValue(roomId, out var chatRoom);

return chatRoom;

}

}

class Program

{

static void Main()

{

var chatRoomManager = ChatRoomManager.Instance;

// Create chat rooms

var room123 = chatRoomManager.CreateChatRoom("Room123");

// Create users

var alice = new User("Alice");

var bob = new User("Bob");

var charlie = new User("Charlie");

// Users join chat rooms

alice.JoinChatRoom(room123);

bob.JoinChatRoom(room123);

charlie.JoinChatRoom(room123);

// Users send messages

alice.SendMessage("Hello, everyone!");

bob.SendMessage("How's it going?");

charlie.SendMessage("Goodbye!");

// Display active users and messages

Console.WriteLine("Active users:");

foreach (var user in room123.GetActiveUsers())

{

Console.WriteLine(user);

}

Console.WriteLine("\nMessage history:");

foreach (var message in room123.GetMessages())

{

Console.WriteLine(message);

}

// Users leave chat rooms

alice.LeaveChatRoom();

bob.LeaveChatRoom();

charlie.LeaveChatRoom();

}

}

}

6. using System;

namespace SatelliteCommandSystem

{

public class Satellite

{

// Satellite's attributes

public string Orientation { get; private set; }

public string SolarPanels { get; private set; }

public int DataCollected { get; private set; }

// Constructor to initialize the satellite

public Satellite()

{

Orientation = "North";

SolarPanels = "Inactive";

DataCollected = 0;

}

// Method to rotate the satellite

public void Rotate(string direction)

{

if (direction == "North" || direction == "South" || direction == "East" || direction == "West")

{

Orientation = direction;

Console.WriteLine($"Orientation set to {Orientation}.");

}

else

{

Console.WriteLine("Invalid direction. Use North, South, East, or West.");

}

}

// Method to activate the solar panels

public void ActivatePanels()

{

SolarPanels = "Active";

Console.WriteLine("Solar panels activated.");

}

// Method to deactivate the solar panels

public void DeactivatePanels()

{

SolarPanels = "Inactive";

Console.WriteLine("Solar panels deactivated.");

}

// Method to collect data

public void CollectData()

{

if (SolarPanels == "Active")

{

DataCollected += 10;

Console.WriteLine($"Data collected: {DataCollected} units.");

}

else

{

Console.WriteLine("Cannot collect data. Solar panels are inactive.");

}

}

// Method to display the current state of the satellite

public void DisplayStatus()

{

Console.WriteLine($"Current Status:");

Console.WriteLine($"Orientation: {Orientation}");

Console.WriteLine($"Solar Panels: {SolarPanels}");

Console.WriteLine($"Data Collected: {DataCollected} units");

}

}

class Program

{

static void Main()

{

// Initialize the satellite

Satellite satellite = new Satellite();

satellite.DisplayStatus();

// Execute commands

satellite.Rotate("South");

satellite.ActivatePanels();

satellite.CollectData();

// Display the final state

satellite.DisplayStatus();

}

}

}

7. #include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#define MAX\_FUEL 100

#define FUEL\_CONSUMPTION\_PER\_SECOND 10

#define INITIAL\_ALTITUDE 0

#define INITIAL\_SPEED 0

#define ALTITUDE\_INCREMENT 10

#define SPEED\_INCREMENT 1000

#define STAGE\_SEPARATION\_ALTITUDE 100

typedef struct {

int fuel;

int altitude;

int speed;

int stage;

int is\_launching;

} Rocket;

void print\_status(Rocket \*rocket) {

printf("Stage: %d, Fuel: %d%%, Altitude: %d km, Speed: %d km/h\n",

rocket->stage, rocket->fuel, rocket->altitude, rocket->speed);

}

void launch\_rocket(Rocket \*rocket) {

rocket->is\_launching = 1;

while (rocket->fuel > 0 && rocket->altitude < STAGE\_SEPARATION\_ALTITUDE) {

rocket->fuel -= FUEL\_CONSUMPTION\_PER\_SECOND;

if (rocket->fuel < 0) rocket->fuel = 0; // Ensure fuel doesn't go negative

rocket->altitude += ALTITUDE\_INCREMENT;

rocket->speed += SPEED\_INCREMENT;

print\_status(rocket);

sleep(1); // Sleep for 1 second to simulate real-time

}

if (rocket->fuel <= 0) {

printf("Mission Failed due to insufficient fuel.\n");

rocket->is\_launching = 0;

return;

}

// Stage separation

if (rocket->altitude >= STAGE\_SEPARATION\_ALTITUDE) {

rocket->stage = 2;

printf("Stage 1 complete. Separating stage. Entering Stage 2.\n");

rocket->altitude = 0; // Reset altitude for the next stage (if needed)

rocket->speed = 0; // Reset speed for the next stage (if needed)

rocket->fuel = 0; // No more fuel in Stage 1

}

}

void fast\_forward(Rocket \*rocket, int seconds) {

if (!rocket->is\_launching) {

printf("Rocket is not yet launched. Please launch the rocket first.\n");

return;

}

for (int i = 0; i < seconds; i++) {

if (rocket->fuel <= 0) {

printf("Mission Failed due to insufficient fuel.\n");

return;

}

rocket->fuel -= FUEL\_CONSUMPTION\_PER\_SECOND;

if (rocket->fuel < 0) rocket->fuel = 0; // Ensure fuel doesn't go negative

rocket->altitude += ALTITUDE\_INCREMENT;

rocket->speed += SPEED\_INCREMENT;

print\_status(rocket);

}

}

int main() {

Rocket rocket = {MAX\_FUEL, INITIAL\_ALTITUDE, INITIAL\_SPEED, 1, 0};

char command[50];

printf("Rocket Launch Simulator\n");

printf("Commands:\n");

printf("start\_checks - Start pre-launch checks\n");

printf("launch - Launch the rocket\n");

printf("fast\_forward X - Fast forward X seconds\n");

while (1) {

printf("> ");

fgets(command, sizeof(command), stdin);

command[strcspn(command, "\n")] = '\0'; // Remove newline character

if (strcmp(command, "start\_checks") == 0) {

printf("All systems are 'Go' for launch.\n");

} else if (strcmp(command, "launch") == 0) {

if (rocket.is\_launching) {

printf("Rocket already launched.\n");

} else {

launch\_rocket(&rocket);

}

} else if (strncmp(command, "fast\_forward", 12) == 0) {

int seconds;

if (sscanf(command + 13, "%d", &seconds) == 1) {

fast\_forward(&rocket, seconds);

} else {

printf("Invalid fast\_forward command. Use 'fast\_forward X', where X is the number of seconds.\n");

}

} else {

printf("Unknown command. Please try again.\n");

}

}

return 0;

}

8. using System;

using System.Collections.Generic;

class VirtualClassroomManager

{

private static Dictionary<string, Classroom> classrooms = new Dictionary<string, Classroom>();

public static void Main()

{

Console.WriteLine("Virtual Classroom Manager");

Console.WriteLine("Commands:");

Console.WriteLine("add\_classroom [ClassName] - Create a new classroom");

Console.WriteLine("add\_student [StudentID] [ClassName] - Enroll a student into a classroom");

Console.WriteLine("schedule\_assignment [ClassName] [AssignmentDetails] - Schedule an assignment");

Console.WriteLine("submit\_assignment [StudentID] [ClassName] [AssignmentDetails] - Submit an assignment");

while (true)

{

Console.Write("> ");

string[] input = Console.ReadLine().Split(new[] { ' ' }, 3);

if (input.Length < 2)

{

Console.WriteLine("Invalid command.");

continue;

}

string command = input[0].ToLower();

string arg1 = input[1];

string arg2 = input.Length > 2 ? input[2] : string.Empty;

switch (command)

{

case "add\_classroom":

AddClassroom(arg1);

break;

case "add\_student":

AddStudent(arg1, arg2);

break;

case "schedule\_assignment":

ScheduleAssignment(arg1, arg2);

break;

case "submit\_assignment":

string[] submitArgs = arg2.Split(new[] { ' ' }, 2);

if (submitArgs.Length == 2)

{

SubmitAssignment(arg1, submitArgs[0], submitArgs[1]);

}

else

{

Console.WriteLine("Invalid submit\_assignment command.");

}

break;

default:

Console.WriteLine("Unknown command.");

break;

}

}

}

private static void AddClassroom(string className)

{

if (classrooms.ContainsKey(className))

{

Console.WriteLine($"Classroom {className} already exists.");

}

else

{

classrooms[className] = new Classroom(className);

Console.WriteLine($"Classroom {className} has been created.");

}

}

private static void AddStudent(string studentId, string className)

{

if (!classrooms.ContainsKey(className))

{

Console.WriteLine($"Classroom {className} does not exist.");

return;

}

Classroom classroom = classrooms[className];

if (classroom.Students.Contains(studentId))

{

Console.WriteLine($"Student {studentId} is already enrolled in {className}.");

}

else

{

classroom.Students.Add(studentId);

Console.WriteLine($"Student {studentId} has been enrolled in {className}.");

}

}

private static void ScheduleAssignment(string className, string assignmentDetails)

{

if (!classrooms.ContainsKey(className))

{

Console.WriteLine($"Classroom {className} does not exist.");

return;

}

Classroom classroom = classrooms[className];

classroom.Assignments.Add(assignmentDetails);

Console.WriteLine($"Assignment for {className} has been scheduled.");

}

private static void SubmitAssignment(string studentId, string className, string assignmentDetails)

{

if (!classrooms.ContainsKey(className))

{

Console.WriteLine($"Classroom {className} does not exist.");

return;

}

Classroom classroom = classrooms[className];

if (!classroom.Students.Contains(studentId))

{

Console.WriteLine($"Student {studentId} is not enrolled in {className}.");

return;

}

if (!classroom.Assignments.Contains(assignmentDetails))

{

Console.WriteLine($"Assignment {assignmentDetails} does not exist for {className}.");

return;

}

Console.WriteLine($"Assignment submitted by Student {studentId} in {className}.");

}

private class Classroom

{

public string Name { get; }

public HashSet<string> Students { get; }

public HashSet<string> Assignments { get; }

public Classroom(string name)

{

Name = name;

Students = new HashSet<string>();

Assignments = new HashSet<string>();

}

}

}