Controlling & Change State of Drone

A Project

Submitted to the - Ahsan kabir

Department of Computer Science and engineering

Bangladesh University of Business and Technology

BY-

Md Nurnobi Hossain intake:34,

ID:19201203059 program- B.Sc in CSE

1.Executive Summary

This code implements a basic drone control system using the state design pattern in C#. The program allows the user to control a drone by entering commands through the console. The drone can perform actions such as landing, takeoff, rotating in different directions (north, south, east, west), and adjusting its speed. The program runs in a loop and continues to accept user commands until the "exit" command is entered.

The state design pattern is utilized to represent the various states of the drone (flying and landed) and encapsulate their specific behaviors. The Drone class serves as the context class, responsible for delegating the input handling to the current state object. Depending on the current state, the appropriate actions are taken or error messages are displayed if invalid commands are entered.

The code provides a framework for expanding the functionality of the drone control system by easily adding new states and extending the behavior of each state.

2. Brief Description of State Design Pattern

The State design pattern is a behavioral design pattern that allows an object to change its behavior dynamically when its internal state changes. It provides a way to manage the behavior of an object by encapsulating the different states as separate classes and delegating the behavior to the current state object.

What is the State Design Pattern? The State design pattern enables an object to alter its behavior at runtime by encapsulating each state as an individual class. The object maintains a reference to the current state object, which handles the requests and changes the behavior accordingly. This approach allows the object to appear as if it has changed its class when its state changes, without modifying its interface.

The State design pattern promotes loose coupling between the object and its states. It abstracts the states into separate classes, making it easier to add new states without modifying existing code. It also improves code readability and maintainability by centralizing state-specific behavior within each state class.

History of the State Design Pattern The State design pattern was first introduced by the

Gang of Four (GoF) in their book "Design Patterns: Elements of Reusable ObjectOriented Software" in 1994. The GoF authors—Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—described the State pattern as one of the 23 fundamental design patterns.

The State design pattern was developed to address situations where an object's behavior changes based on its internal state. By encapsulating the behavior into separate state classes, the pattern allows for easy extensibility, maintainability, and code reusability.

Since its introduction, the State design pattern has been widely adopted in various software development contexts. It is commonly used in scenarios involving finite state machines, where an object's behavior depends on its current state and transitions between states based on external events or actions.

The State design pattern provides a flexible and scalable solution for managing statespecific behavior, making it a valuable tool in software design and development.

3. Implementation

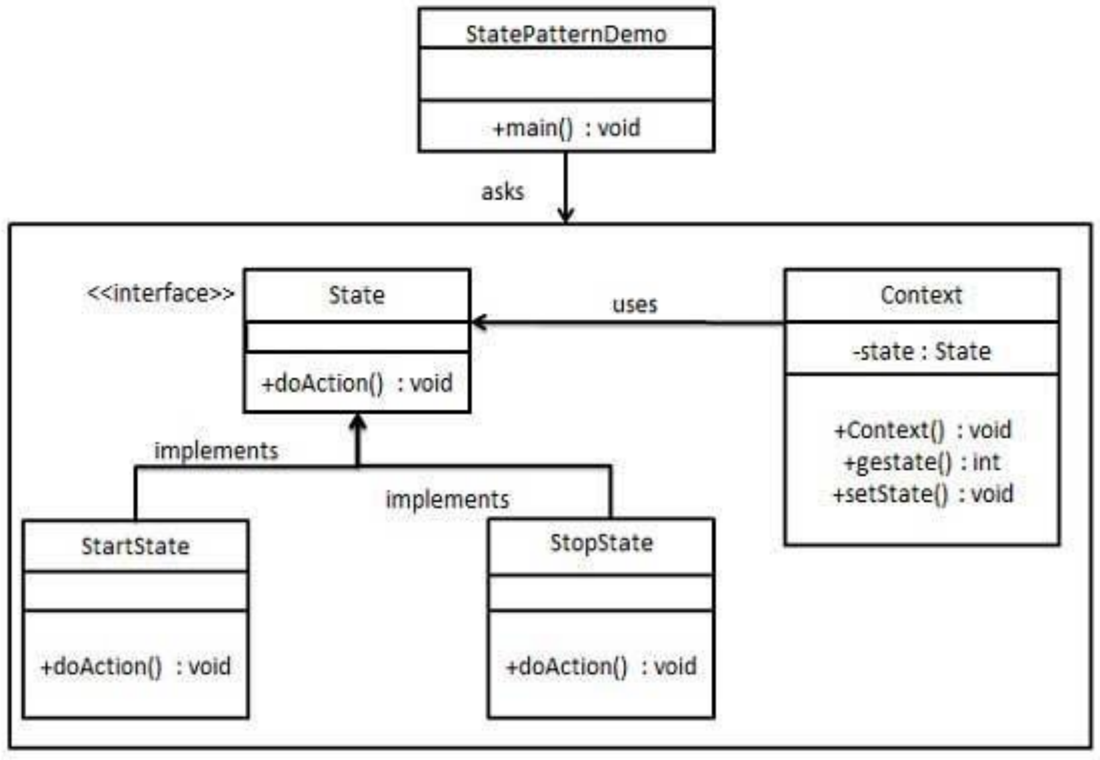
The implementation of the code involves defining the state interface, creating concrete state classes, implementing the drone class, and handling user input. Here's a breakdown of the implementation steps:

1. Define the State Interface:
    - Create the `IDroneState` interface that defines the contract for the different states of the drone. It includes the `HandleInput` method that takes the drone instance and user input as parameters.
2. Implement Concrete State Classes:
    - Implement the concrete state classes that represent the drone states. In this code, we have `FlyingState` and `LandedState` classes that implement the `IDroneState` interface. Each class implements the `HandleInput` method with specific behaviors based on the current state.
3. Create the Drone Class:
    - Define the `Drone` class that maintains the current state of the drone and handles state transitions. It has a `ChangeState` method to update the current state and a `HandleInput` method that delegates input handling to the current state object.
4. Implement Input Handling:
    - In the `Main` method of the `Program` class, create an instance of the `Drone` class. Use a loop to continuously prompt the user for input commands. Within the loop,

call the `HandleInput` method of the `Drone` instance, passing the user input as a parameter.

That's the general implementation structure of the code. You can copy and run the provided code in a C# environment, such as Visual Studio, to test and interact with the drone control system. Feel free to modify or enhance the code according to your specific requirements and add any additional functionality you may need.

- **UML Diagram**: The UML diagram for the car state tracking system using the State design pattern would illustrate the relationships between the classes and their interactions. It would include the Car class, State enumeration, and Action enumeration, along with their associations and dependencies.



- **Description Step by Step:**

  - The program starts by creating an instance of the `Drone` class and setting its initial state to `LandedState`.
  - The program enters a loop that prompts the user to enter a command.
  - The user enters a command (e.g., "takeoff").

- The `HandleInput` method of the `Drone` class is called, passing the user input as a parameter.
- The `Drone` class delegates the input handling to the current state object (in this case, `LandedState`).
- The `HandleInput` method of the `LandedState` class is invoked, and it matches the command "takeoff".
- The `LandedState` class outputs "Taking off..." to the console.
- The `LandedState` class calls the `ChangeState` method of the `Drone` class, transitioning the drone to the `FlyingState`.
- The loop continues, prompting the user for the next command.
- The user enters a command (e.g., "north").
- The `HandleInput` method of the `Drone` class is called again, passing the user input.
- The `Drone` class delegates the input handling to the current state object (in this case, `FlyingState`).
- The `HandleInput` method of the `FlyingState` class is invoked, and it matches the command "north".
- The `FlyingState` class outputs "Rotating north..." to the console.
- The loop continues, and this process repeats until the user enters "exit" as the command.
- Once the user enters "exit", the loop exits, and the program terminates.

Throughout the execution, the program uses the state design pattern to handle the different states of the drone and dynamically adjust its behavior based on the current state. The user's commands are interpreted and processed differently depending on whether the drone is in the "flying" or "landed" state.

4. Execution

When we will execute the code, it will start the drone control system and display a prompt in the console, asking you to enter a command. Here's a step-by-step execution flow:

1. The program starts, and the Main method is called.
2. An instance of the Drone class is created, and its initial state is set to LandedState.
3. The program enters a loop that repeatedly prompts you to enter a command.
4. You enter a command (e.g., "takeoff") and press enter.
5. The HandleInput method of the Drone class is called, passing the entered command as a parameter.
6. The Drone class delegates the command handling to the current state object (LandedState in this case).

7. The HandleInput method of the LandedState class is invoked, and it recognizes the command "takeoff".
8. The LandedState class outputs "Taking off..." to the console.
9. The LandedState class calls the ChangeState method of the Drone class to transition to the FlyingState.
10. The program continues to the next iteration of the loop.
11. You enter another command (e.g., "north") and press enter.
12. The HandleInput method of the Drone class is called again.
13. This time, the Drone class delegates the command handling to the current state object (FlyingState).
14. The HandleInput method of the FlyingState class is invoked, and it recognizes the command "north".
15. The FlyingState class outputs "Rotating north..." to the console.
16. The program continues to the next iteration of the loop, prompting you for the next command.
17. You can continue entering commands, such as "land", "speed 2", "east", etc., and the program will execute the corresponding actions based on the current state of the drone.
18. The loop will continue until you enter "exit" as the command.
19. When "exit" is entered, the loop breaks, and the program terminates.

Throughout the execution, the code follows the state design pattern to manage the behavior of the drone based on its current state. The input commands are interpreted and processed accordingly, producing the desired actions and output messages in the console.

5. Program Output

The output of the program will depend on the commands you enter. Here's an example of the program's output for a sequence of commands:

```
Enter a command (land, takeoff, north, south, east, west, speed [1-3], exit):
takeoff
Taking off...
Enter a command (land, takeoff, north, south, east, west, speed [1-3], exit):
north
Rotating north...
Enter a command (land, takeoff, north, south, east, west, speed [1-3], exit):
speed 2
Setting the drone speed to 2.
Enter a command (land, takeoff, north, south, east, west, speed [1-3], exit):
land
Landing the drone...
Enter a command (land, takeoff, north, south, east, west, speed [1-3], exit):
exit
```

The program prompts you to enter a command each time and displays the corresponding output based on the entered command and the current state of the drone. For example, when you enter "takeoff", the program outputs "Taking off...". When you enter "north", the program outputs "Rotating north...". When you enter "land", the program outputs "Landing the drone...". And when you enter "exit", the program terminates.

The specific output messages and actions will vary based on the commands you enter and the logic you implement within the different state classes. You can modify the code and add more specific behaviors to customize the output based on your requirements.

6.Code

Program.cs

```
using System;

// Step 1: Define the State Interface
interface IDroneState
{
    void HandleInput(Drone drone, string input);
}

// Step 2: Implement Concrete State Classes
class FlyingState : IDroneState
{
    public void HandleInput(Drone drone, string input)
    {
        switch (input)
        {
            case "land":
                Console.WriteLine("Landing the drone...");
                drone.ChangeState(new LandedState());
                break;

            case "takeoff":
                Console.WriteLine("The drone is already flying.");
                break;

            case "north":
                Console.WriteLine("Rotating north...");
                // Implement the rotation logic here
                break;

            case "south":
                Console.WriteLine("Rotating south...");
                // Implement the rotation logic here
                break;

            case "east":
                Console.WriteLine("Rotating east...");
                // Implement the rotation logic here
                break;

            case "west":
                Console.WriteLine("Rotating west...");
                // Implement the rotation logic here
                break;

            case "speed 1":
                Console.WriteLine("Setting the drone speed to 1.");
                // Implement the speed adjustment logic here
                break;

            case "speed 2":
                Console.WriteLine("Setting the drone speed to 2.");
                // Implement the speed adjustment logic here
                break;

            case "speed 3":
                Console.WriteLine("Setting the drone speed to 3.");
                // Implement the speed adjustment logic here
                break;

            default:
                Console.WriteLine("Invalid command.");
                break;
        }
    }
}

class LandedState : IDroneState
{
    public void HandleInput(Drone drone, string input)
    {
        switch (input)
        {
            case "land":
                Console.WriteLine("The drone is already landed.");
                break;

            case "takeoff":
                Console.WriteLine("Taking off...");
                drone.ChangeState(new FlyingState());
                break;

            case "north":
            case "south":
            case "east":
            case "west":
                Console.WriteLine("The drone needs to be flying to rotate.");
                break;

            case "speed 1":
            case "speed 2":
            case "speed 3":
                Console.WriteLine("The drone needs to be flying to set the speed.");
                break;

            default:
                Console.WriteLine("Invalid command.");
                break;
        }
    }
}

// Step 3: Create a Drone Class
class Drone
{
    private IDroneState currentState;

    public Drone()
    {
        currentState = new LandedState();
    }

    public void ChangeState(IDroneState newState)
    {
        currentState = newState;
    }

    public void HandleInput(string input)
    {
            currentState.HandleInput(this, input);
        }
    }

    // Step 4: Implement Input Handling
    class Program
    {
        static void Main()
        {
            Drone drone = new Drone();

            string input = "";
            while (input != "exit")
            {
                Console.WriteLine("Enter a command (land, takeoff, north, south, east, west, speed
    [1-3], exit):");
                input = Console.ReadLine().ToLower();

                drone.HandleInput(input);
            }
        }
    }
```

Conclusion :

In conclusion, the provided C# code implements a basic drone control system using the state design pattern. It allows users to control a drone by entering commands through the console. The program utilizes the state pattern to encapsulate the different states of the drone (flying and landed) and dynamically adjust its behavior based on the current state.

The code demonstrates a modular and extensible design that separates the state-specific logic into individual state classes. This makes it easier to add new states and modify the behavior of the drone without impacting the overall structure of the program.

By following the state design pattern, the code promotes code reusability, maintainability, and flexibility. It provides a foundation for further enhancements and the addition of more advanced functionalities to the drone control system.

Overall, this code serves as a starting point for building a drone control application and showcases the benefits of utilizing the state design pattern for managing the behavior of complex systems with multiple states.