Research project: Implementing the State Pattern in C# language.

COS20007

XUAN DAT LE - 103487949

Table of Contents

Abstract

This research project focuses on the implementation of the State Pattern in the C# programming language. The State Pattern is a behavioral design pattern that allows an object to alter its behavior when its internal state changes. The objective of this project is to explore the practical application of the State Pattern in real-world scenarios and evaluate its effectiveness in enhancing code flexibility, modularity, and maintainability.

Introduction

The State Pattern is a widely used design pattern in software development that allows objects to dynamically change their behavior based on internal states. By encapsulating states in separate classes and enabling transitions between them, the State Pattern enhances code flexibility, modularity, and maintainability. This research project focuses on implementing the State Pattern in C# in the HD Program that I have created myself to explore its practical application and evaluate its effectiveness in software design. The project involves developing a sample project, analyzing its design, performance, and comparisons with alternative approaches. The outcomes will contribute to knowledge on design patterns, inform decision-making for developers, and inspire further exploration in software development.

Method

The literature review was conducted, encompassing an extensive examination of relevant academic papers, books, articles, and online resources that focused on the State Pattern. The findings from the literature review were analyzed and synthesized to gain insights into the principles, benefits, and limitations of the State Pattern. Furthermore, real-world use cases and examples in the HD program created by me where the State Pattern had been successfully applied were identified, providing valuable insights into its practical applications in software development.

1. Scenario

Considering the scenario: In a smart home automation system, multiple devices have different operational modes. Initially, managing the behavior of each device based on its mode involved numerous if-else statements, leading to code complexity and maintenance challenges.

To address this, the state pattern was introduced. The system now utilizes a context class that maintains the current state object for each device. Each device has its own state class, encapsulating the behavior for each mode. This implementation simplifies mode transitions and improves code organization, resulting in a more manageable and scalable smart home system.

```
public SmartHomeDevice(string initialMode)
{
    this.mode = initialMode;
}

public void ChangeMode(string newMode)
{
    if (newMode == "On")
    {
        // Perform actions for switching to "On" mode
        // ...
        this.mode = newMode;
    }
    else if (newMode == "Off")
    {
        // Perform actions for switching to "Off" mode
        // ...
        this.mode = newMode;
    }
    else if (newMode == "Eco")
    {
        // Perform actions for switching to "Eco" mode
        // ...
        this.mode = newMode;
    }
    // More if-else statements for additional modes
}

public void PerformOperation()
{
    if (this.mode == "On")
    {
        // Perform actions for "On" mode
        // ...
    }
    else if (this.mode == "Off")
    {
        // Perform actions for "Off" mode
        // ...
    }
    else if (this.mode == "Eco")
    {
        // Perform actions for "Eco" mode
```

Figure 1: Code implementation without the State Pattern where multiple if-else statements are used.

2. Definition of the State Pattern

The state pattern belongs to the category of behavioral design patterns. It is employed when an object's behavior undergoes changes depending on its internal state. When we need to alter an object's behavior based on its state, we can introduce a state variable within the object and utilize an if-else condition block to execute various actions according to the state. The state pattern offers a structured and loosely coupled approach to achieve this functionality by implementing the Context and State components.

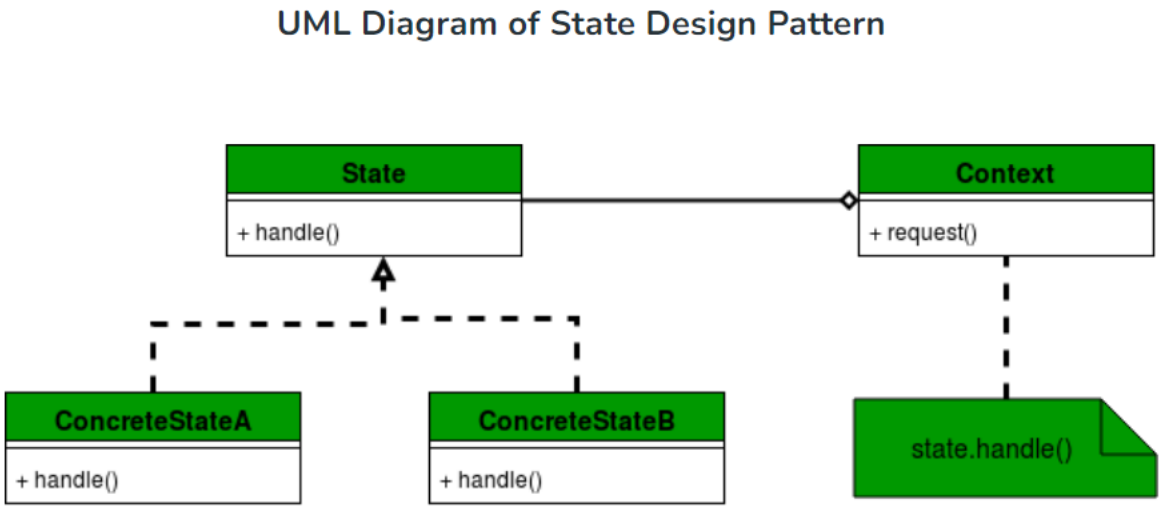


Figure 2: UML Diagram of State Design Pattern

In the UML Diagram above, the **Context** defines an interface for clients to interact. It maintains references to concrete state objects which may be used to define the current state of objects. On top of that, the **State** defines interface for declaring what each concrete state should do. Finally, the **Concrete State** provides the implementation for methods defined in State.

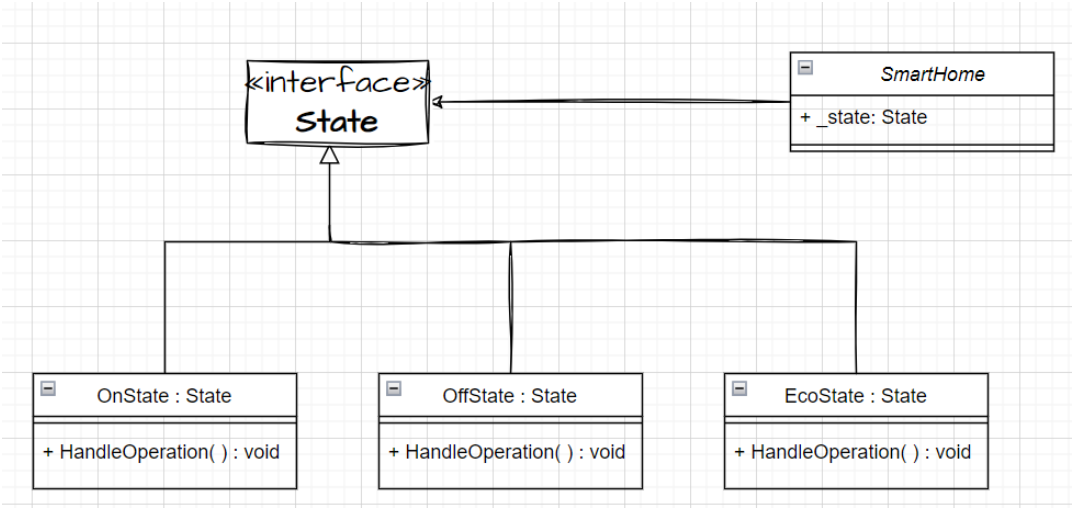


Figure 3: State Pattern implementation for the opening scenario

3. Advantages of the State Design Pattern

The State pattern offers several benefits when implementing polymorphic behavior and adding support for additional states. It eliminates the need for complex if/else or switch/case statements by allowing an object's behavior to change dynamically based on its state.

In the Smart Home example, the State pattern offers similar benefits. Instead of relying on a single class with conditional logic, the State pattern allows the behavior to be encapsulated within separate ConcreteState classes. This approach enhances code cohesion and readability by dividing the behavior into distinct states with clear and descriptive names, such as "OnState," "OffState," and "EcoState." This modular structure simplifies comprehension and makes the code more maintainable. By leveraging the State pattern, the smart home system gains flexibility and modularity in managing the behavior of devices based on their varying states.

4. Disadvantages of the State Pattern

While the State pattern offers advantages, it also has some drawbacks.

First, it can increase code complexity by introducing multiple state classes and transitions, making the system harder to understand and maintain. The pattern may lead to a larger class hierarchy and impact performance due to the increased number of classes.

Second, it may not scale well with many states or complex transitions, making management challenging. The coupling between the context and state classes can make modifications and stability maintenance more cumbersome.

Additionally, there can be a slight performance overhead due to the indirection involved in dispatching behavior based on the current state.

Considering these factors is crucial when deciding whether to use the State pattern in performance-critical or complex systems.

5. Real-life implementation

In the HD custom program, in the original draft of the game, the main game was written in one big file, with bunch of if-else syntaxes to moving between states. This leads to difficulty in understanding the program thoroughly for reviewers as well as struggle in maintenance.

```
            SwinGame.ProcessEvents();
            //clearing screen to white
            SwinGame.ClearScreen(Color.White);
            SwinGame.DrawText("In this game, you are a black block.", Color.Black, optimusFont, 50, 50);
            SwinGame.DrawText("Basically you just shoot anything that's not you.", Color.Black, optimusFont, 50, 100);
            SwinGame.DrawText("If you touch other blocks, you lose HP equal to theirs", Color.Black, optimusFont, 50, 150);
            SwinGame.DrawText("ESC - Pause", Color.Black, optimusFont, 50, 200);
            SwinGame.DrawText("W A S D - Move", Color.Black, optimusFont, 50, 250);
            SwinGame.DrawText("Arrow keys - shoot at a direction", Color.Black, optimusFont, 50, 300);
            SwinGame.DrawText("R - reload", Color.Black, optimusFont, 50, 350);
            SwinGame.DrawText("W/A/S/D + LEFT SHIFT - teleport", Color.Black, optimusFont, 50, 400);
            SwinGame.DrawText("The range of teleportation is the green square around you", Color.Black, optimusFont, 50, 450);
            SwinGame.DrawText("Teleportation cooldown: 1.5s | Reload time: 1s", Color.Black, optimusFont, 50, 500);
            SwinGame.DrawText("Press H again or ESC to return to the main screen", Color.Black, optimusFont, 50, 550);
            //drawing things out
            SwinGame.RefreshScreen(60);
            if (SwinGame.KeyTyped(KeyCode.HKey) || SwinGame.KeyTyped(KeyCode.EscapeKey))
            {
                SwinGame.ClearScreen(Color.White);
                break;
            }
        }
    }
    if (SwinGame.KeyTyped(KeyCode.Num1Key))
    {
        difficulty = 1;
        gamestart = true;
    }
    if (SwinGame.KeyTyped(KeyCode.Num2Key))
    {
        difficulty = 2;
        gamestart = true;
    }
    if (SwinGame.KeyTyped(KeyCode.Num3Key))
    {
        difficulty = 3;
        gamestart = true;
    }
    //renewing objects
    p = new Player();
    horde = new EnemyHorde(difficulty, p);
    ReloadTimer = new Timer();
    StageTransitionTimer = new Timer();
    enemyEntities = new EntitiesGroup();
    playerEntity = new EntitiesGroup();
    bulletEntities = new EntitiesGroup();
    powerUpEntities = new EntitiesGroup();
    powerUpSpawner = new PowerUpSpawner(p, new Bitmap[] { freeze, speedBoost }, enemyEntities);
    playerEntity.AddObject(p);
    foreach (Enemy e in horde.EnemyList)
        enemyEntities.AddObject(e);
}
```

Figure 5: Original Main game

Realizing this issue, the State Pattern was implemented to counter all the stated drawbacks, here is the UML of the State Pattern diagram applied in my program context.

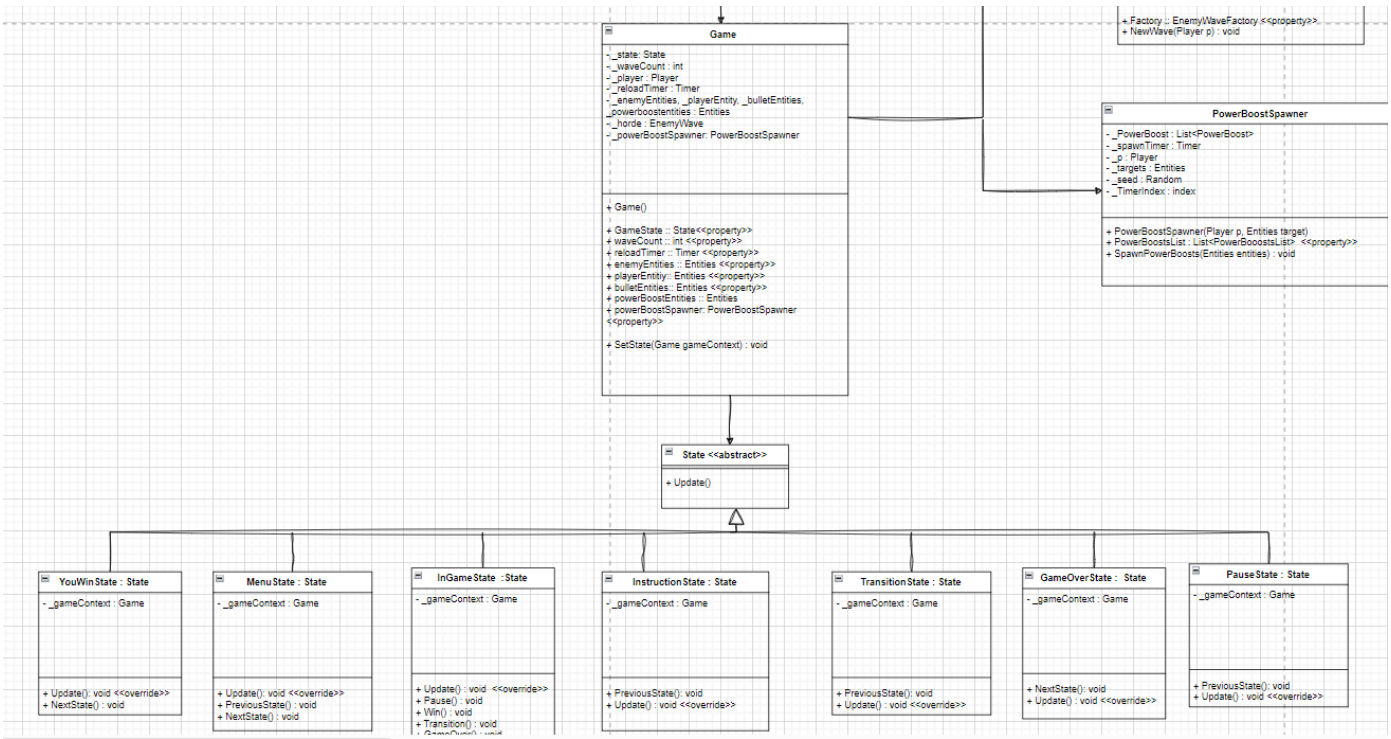


Figure 6: State Pattern in HD Program

With the State Pattern implemented, the Main class now get reduced to just a few lines of code:

```
0 references
public class Program
{
    0 references
    public static void Main()
    {
        //Open the game window
        Game gameContext = new Game();
        SplashKit.OpenWindow("Shooter", 1000, 650);

        //Run the game loop
        gameContext.Run();
    }
}
```

Figure 7: Main class after implement State Pattern

In experiment to test the performance of the State Pattern, I have built a single program which implemented the design pattern, and that version of the program without the pattern and test the runtime using ***Stopwatch*** variable in C#.

Here is the class diagram of the first program, representing the traffic light system:

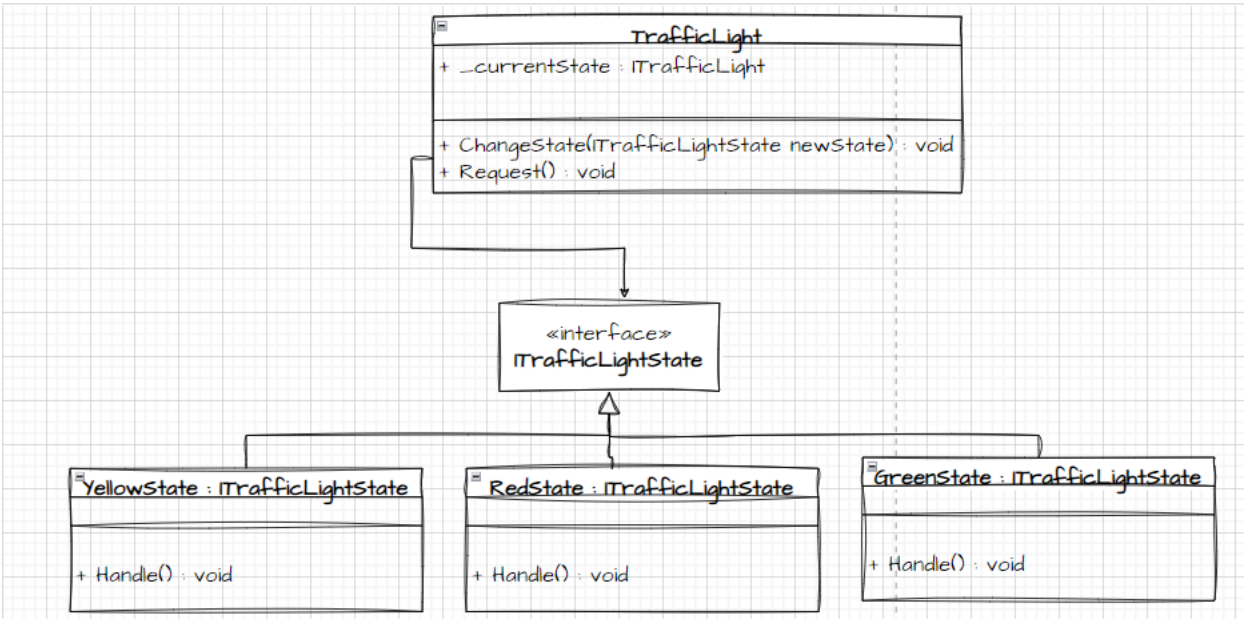


Figure 8: UML Diagram of the first simple program reflecting the traffic light system.

In that diagram, the Handle () method in each state will print out the lighting state, while the Request () method will ask the _currentState field to trigger its Handle () method.

With that design, the difference can easily be observed with the second program's diagram below:

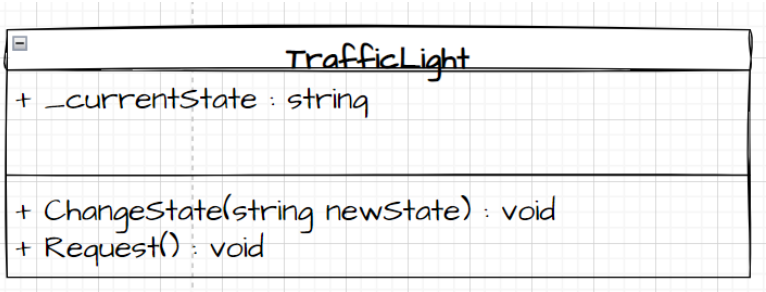


Figure 9: The UML diagram of the non-state design, with only one class

```
// Interface for the traffic light states
public interface ITrafficLightState
{
    void Handle();
}

// Concrete implementation of the "Green" state
public class GreenState : ITrafficLightState
{
    public void Handle()
    {
        Console.WriteLine("Traffic light is green. Go!");
    }
}

// Concrete implementation of the "Yellow" state
public class YellowState : ITrafficLightState
{
    public void Handle()
    {
        Console.WriteLine("Traffic light is yellow. Prepare to stop.");
    }
}

// Concrete implementation of the "Red" state
public class RedState : ITrafficLightState
{
    public void Handle()
    {
        Console.WriteLine("Traffic light is red. Stop!");
    }
}

// Context class that maintains the current state of the traffic light
public class TrafficLight
{
    private ITrafficLightState currentState;

    public TrafficLight()
    {
        currentState = new GreenState();
    }

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

    public void Request()
    {
        currentState.Handle();
    }
}
```

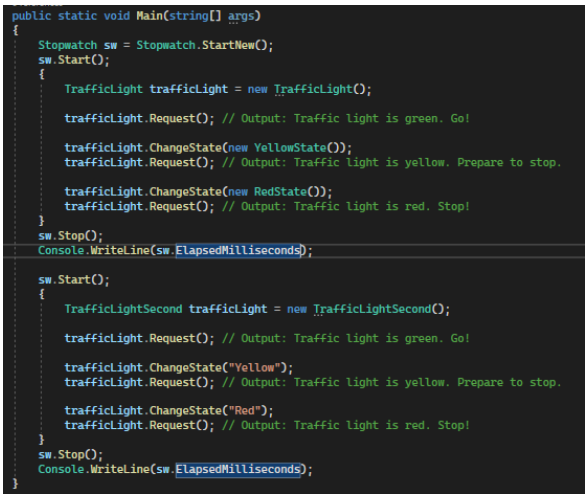
```
1 reference
public TrafficLightSecond()
{
    currentState = "Green";
}

2 references
public void ChangeState(string newState)
{
    currentState = newState;
}

3 references
public void Request()
{
    switch (currentState)
    {
        case "Green":
            Console.WriteLine("Traffic light is green. Go!");
            break;
        case "Yellow":
            Console.WriteLine("Traffic light is yellow. Prepare to stop.");
            break;
        case "Red":
            Console.WriteLine("Traffic light is red. Stop!");
            break;
        default:
            Console.WriteLine("Invalid state.");
            break;
    }
}
```

Figure 10: First program (left image) and Second program (right image)

After drafting 2 programs, the test performance will be conducted using Stopwatch variable as mentioned above:



```
public static void Main(string[] args)
{
    Stopwatch sw = Stopwatch.StartNew();
    sw.Start();
    {
        TrafficLight trafficLight = new TrafficLight();

        trafficLight.Request(); // Output: Traffic light is green. Go!

        trafficLight.ChangeState(new YellowState());
        trafficLight.Request(); // Output: Traffic light is yellow. Prepare to stop.

        trafficLight.ChangeState(new RedState());
        trafficLight.Request(); // Output: Traffic light is red. Stop!
    }
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);

    sw.Start();
    {
        TrafficLightSecond trafficLight = new TrafficLightSecond();

        trafficLight.Request(); // Output: Traffic light is green. Go!

        trafficLight.ChangeState("Yellow");
        trafficLight.Request(); // Output: Traffic light is yellow. Prepare to stop.

        trafficLight.ChangeState("Red");
        trafficLight.Request(); // Output: Traffic light is red. Stop!
    }
    sw.Stop();
    Console.WriteLine(sw.ElapsedMilliseconds);
}
```

Figure 11: Test performance

Results

This section would lead reader through the result of the test above, as well as highlighting the main advantages and disadvantages of 2 design: state and non-state pattern.

	With State Pattern	***Without State Pattern***
Modular and flexible	✔	✘
Easy addition of states	✔	✘
Clear separation of behavior	✔	✘
Improved code readability	✔	✘
Increased complexity	✘	✔
Higher number of classes	✘	✔
Limited scalability	✘	✔
Coupling between context and state	✘	✔
Reduced efficiency	✘	✔

The ticks indicate that an advantage, while the crosses indicate the opposite. With the table above, both the drawbacks and benefits of utilizing the State Pattern were highlighted.

Here is also the runtime of 2 programs in milliseconds.

Attempts / Test Object	First Traffic Program (In milliseconds.)	Second Traffic Program (In milliseconds.)
1.	7	10
2.	8	9
3.	10	10
4.	7	10
5.	12	15

From the table above, we can see that the first program with the State Pattern perform slightly better than the second one without the pattern. Note that this is just a simple program; therefore, the impact of the State Pattern can greatly overperform in large-scale application.

Conclusion

In conclusion, the research project examined the implementation and evaluation of the State pattern in software design. Through a comprehensive analysis of its application, the State pattern demonstrated several advantages, such as modular and flexible code, easy addition of states, and improved code readability. However, potential disadvantages include increased complexity, a higher number of classes, limited scalability, coupling between context and state, and reduced efficiency. Overall, the State pattern provides an effective solution for managing object behavior based on internal states, offering flexibility and maintainability. Future research can explore optimizations and alternative patterns to address its limitations, while comparative studies can provide valuable insights for pattern selection. The State pattern contributes to code modularity, flexibility, and robust system development.

Reference

1. Stack Overflow. (2012). How can I do a performance test in C#?
Retrieved from https://stackoverflow.com/questions/8486454/how-can-i-do-test-for-performance-in-c-sharp.

2. GeeksforGeeks. (n.d.). State Design Pattern.
Retrieved from https://www.geeksforgeeks.org/state-design-pattern/.

3. "The State design pattern - Problem, Solution, and Applicability". (n.d.). w3sDesign.com. Retrieved August 12, 2017.