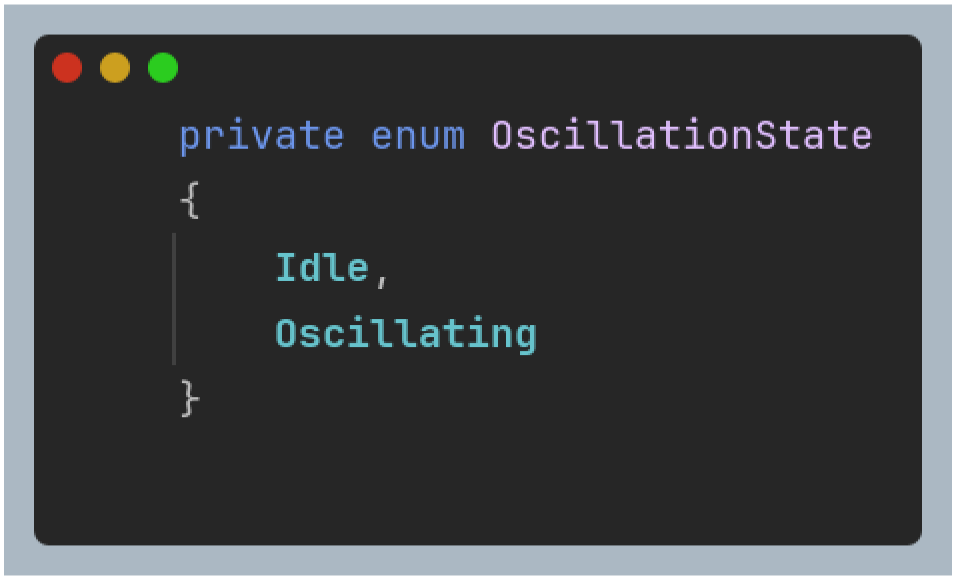
State Pattern:

I have used “State Pattern” in my oscillator class.

In the code:

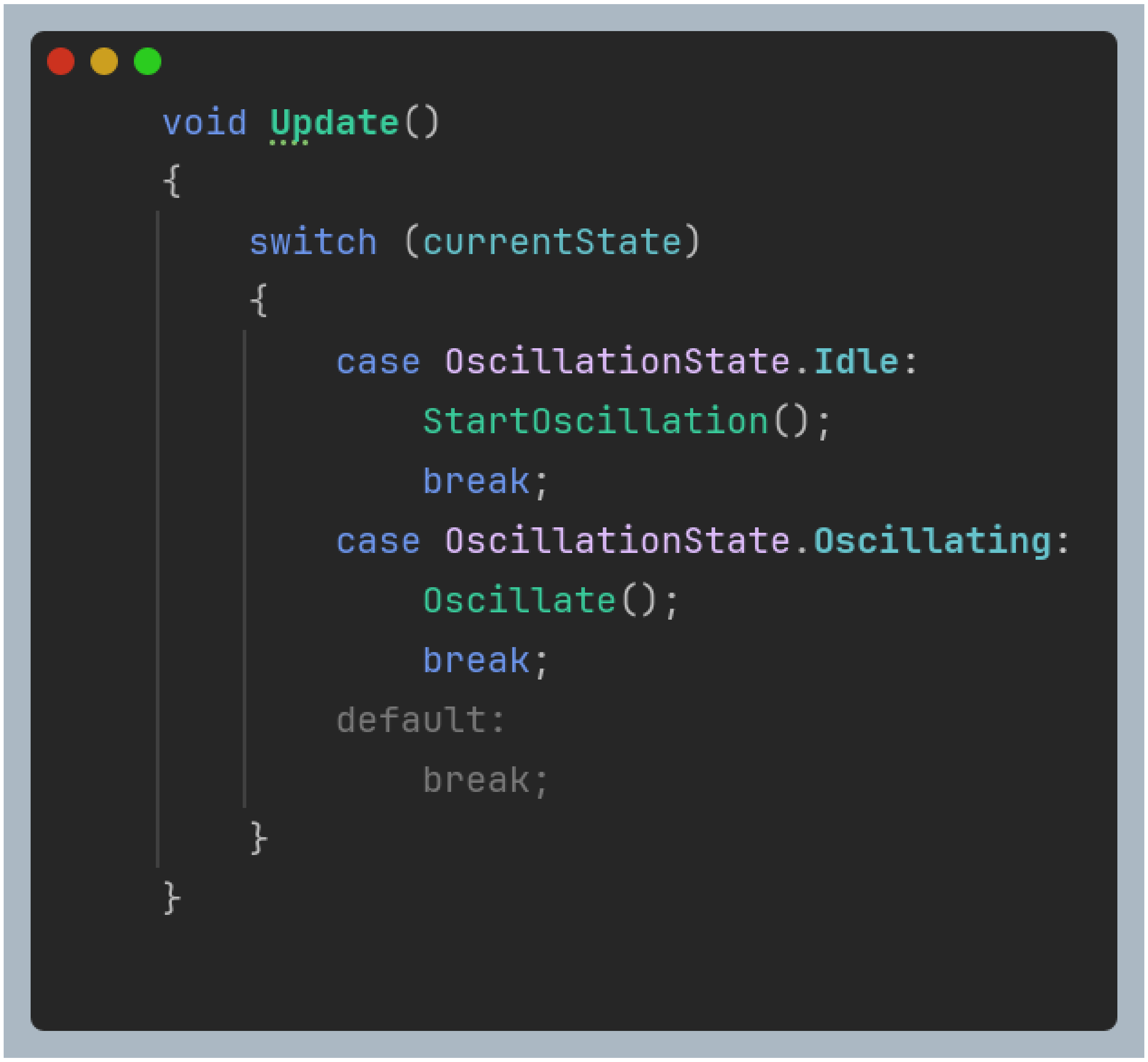
The Oscillator class has two states: Idle and Oscillating, represented by the OscillationState enum.



```
private enum OscillationState
{
    Idle,
    Oscillating
}
```

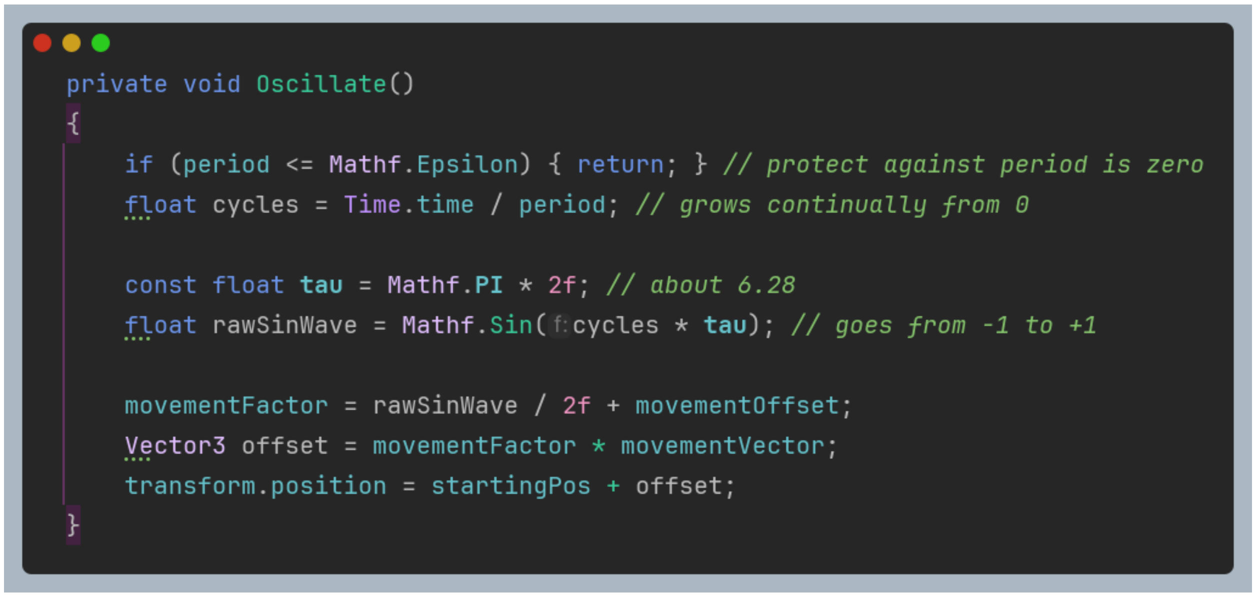
The behavior of the Oscillator class changes based on its current state:

- In the Idle state, it waits for a trigger to start oscillating.

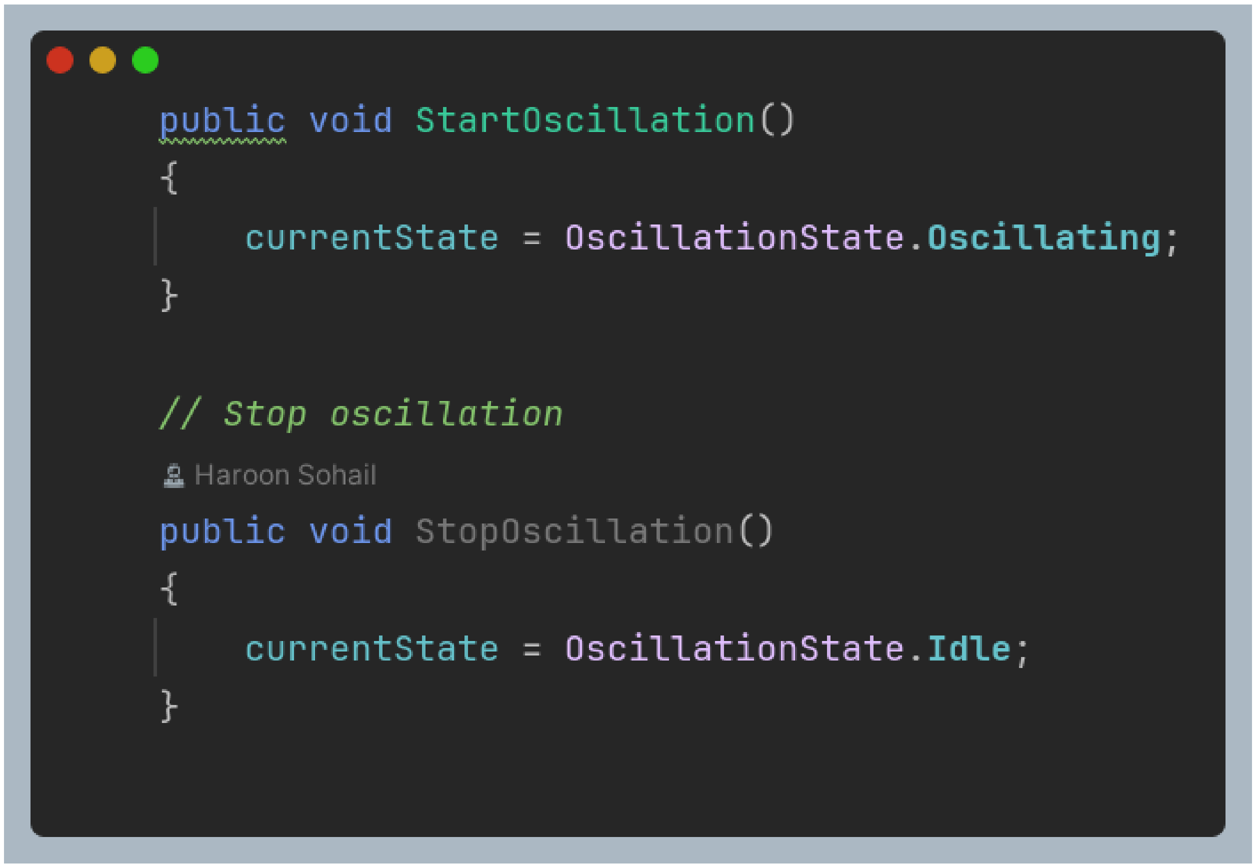


```
void Update()
{
    switch (currentState)
    {
        case OscillationState.Idle:
            StartOscillation();
            break;
        case OscillationState.Oscillating:
            Oscillate();
            break;
        default:
            break;
    }
}
```

- In the Oscillating state, it continuously oscillates its position based on the sine wave calculation in the Oscillate() method.


```
private void Oscillate()
{
    if (period <= Mathf.Epsilon) { return; } // protect against period is zero
    float cycles = Time.time / period; // grows continually from 0

    const float tau = Mathf.PI * 2f; // about 6.28
    float rawSinWave = Mathf.Sin( f: cycles * tau); // goes from -1 to +1

    movementFactor = rawSinWave / 2f + movementOffset;
    Vector3 offset = movementFactor * movementVector;
    transform.position = startingPos + offset;
}
```

- The StartOscillation() and StopOscillation() methods transition the Oscillator between its states.


```
public void StartOscillation()
{
    currentState = OscillationState.Oscillating;
}


// Stop oscillation
Haroon Sohail
public void StopOscillation()
{
    currentState = OscillationState.Idle;
}
```

Advantages:

Modularity: The State Pattern promotes modularity by encapsulating state-specific behavior into separate state objects. This makes it easier to manage and understand the behavior of each state.

Flexibility: It allows for easy addition or modification of states without affecting the core functionality of the class. New states can be added by simply creating new state classes.

Cleaner Code: By separating state-specific behavior, the code becomes more organized and readable. Each state class focuses on a specific aspect of behavior, leading to clearer and more maintainable code.

Dynamic Behavior: The State Pattern enables dynamic behavior changes at runtime. The context class can switch between states based on certain conditions or triggers, allowing for adaptive behavior.

Disadvantages:

Increased Complexity: Implementing the State Pattern may introduce additional complexity, especially if there are many states or if state transitions are complex.

Potential Overhead: Managing state objects and transitions may introduce some overhead, especially if there are frequent state changes.

Potential Tight Coupling: There is a risk of tight coupling between the context class and its state classes if not implemented carefully. Changes in one state class may affect others, leading to maintenance issues.

Alternative Design Patterns:

Strategy Pattern: While the Strategy Pattern is similar to the State Pattern, it focuses on defining a family of algorithms and encapsulating each algorithm into separate classes. In this context, if the oscillation behavior is represented by different algorithms (e.g., sine wave, cosine wave), the Strategy Pattern could be used to encapsulate each algorithm into its own strategy class.

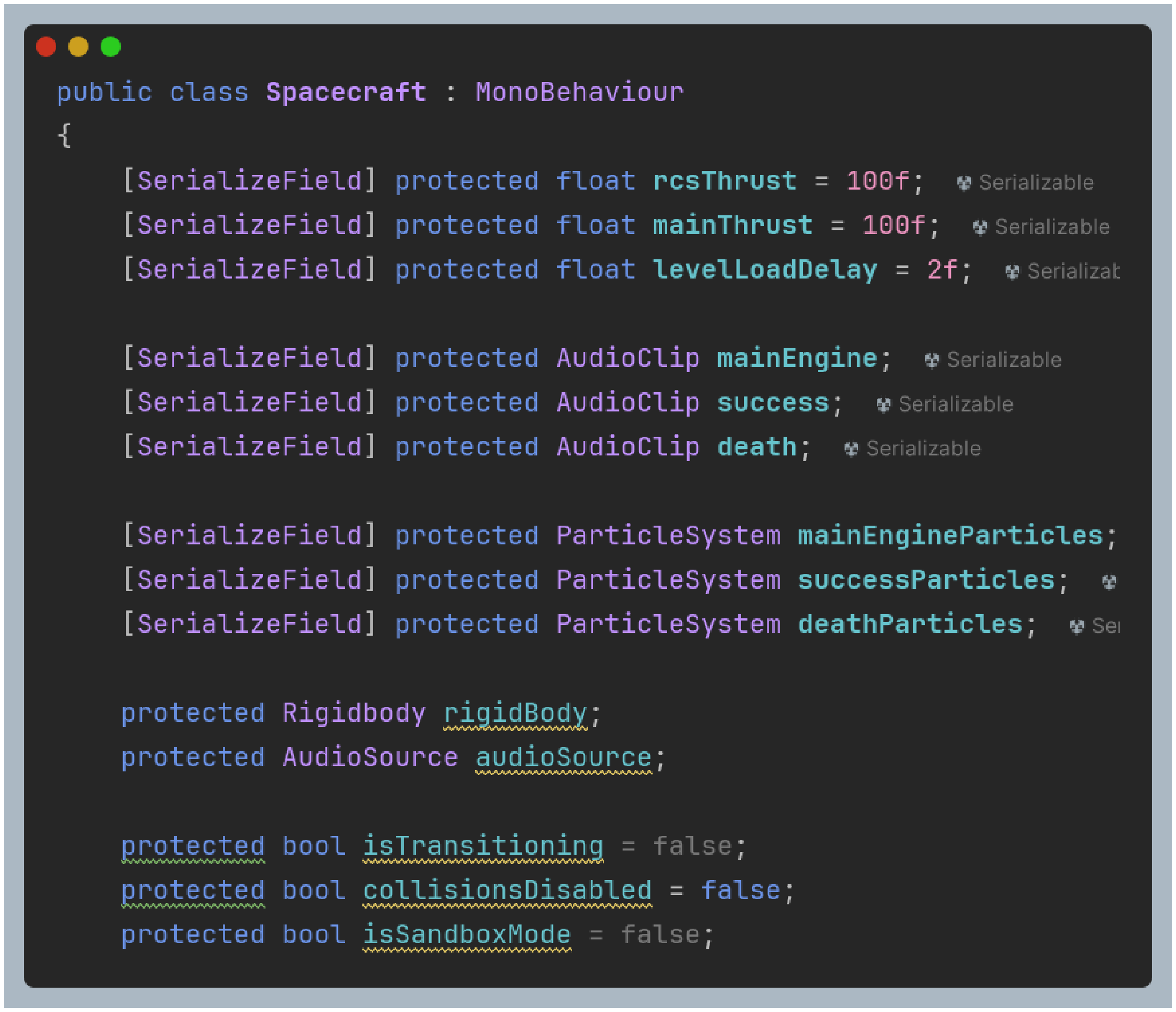
Command Pattern: The Command Pattern could be used to represent oscillation commands as objects. Each command object encapsulates a specific oscillation action, and the Oscillator class can execute these commands based on certain triggers or conditions. This could provide more flexibility in managing oscillation behavior and allow for easy addition or modification of commands.

Observer Pattern: If the oscillation behavior needs to be observed by other objects in the scene, the Observer Pattern could be used. The Oscillator class would act as the subject, and other objects interested in its state changes (e.g., start or stop oscillation) would register as observers. This pattern would facilitate loose coupling between the Oscillator class and its observers.

Why State Pattern was Chosen:

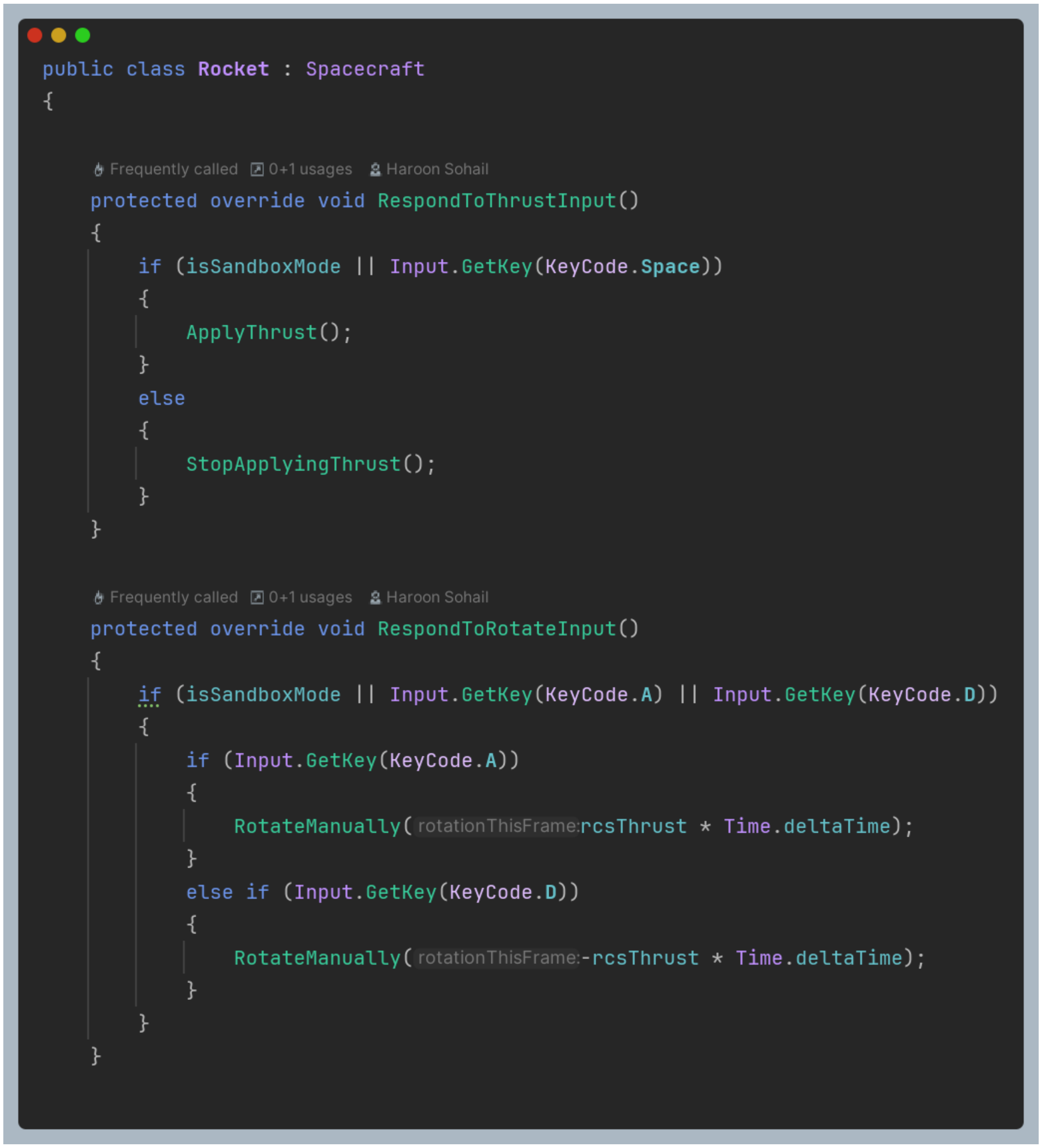
The State Pattern was chosen for this context because it effectively encapsulates the different oscillation states and their behaviors. It provides a clear and organized way to manage state-specific behavior, making the code more modular and maintainable. Additionally, the dynamic behavior changes at runtime, such as starting or stopping oscillation, align well with the capabilities of the State Pattern. Overall, the State Pattern offers a suitable solution for managing the oscillation behavior of the Oscillator class.

Subclass Sandbox:



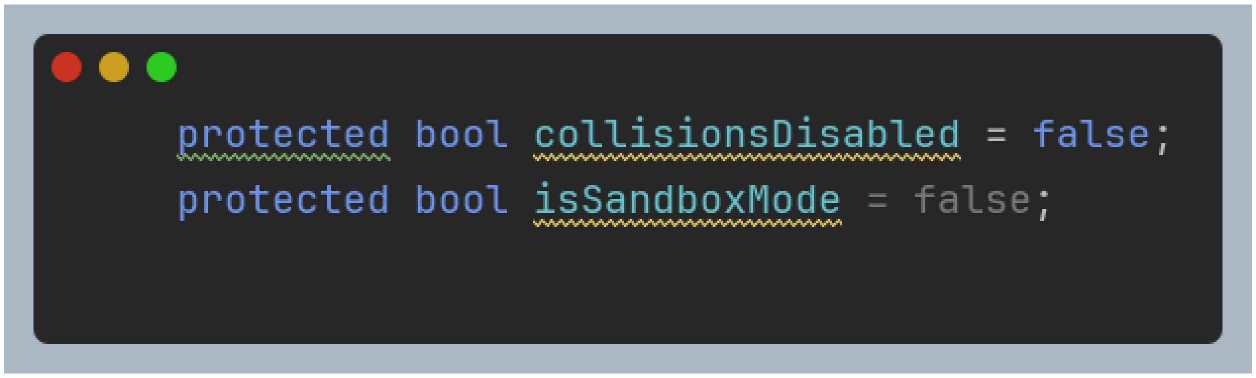
```
public class Spacecraft : MonoBehaviour
{
    [SerializeField] protected float rcsThrust = 100f;   Serializable
    [SerializeField] protected float mainThrust = 100f;   Serializable
    [SerializeField] protected float levelLoadDelay = 2f;   Serializa

    [SerializeField] protected AudioClip mainEngine;   Serializable
    [SerializeField] protected AudioClip success;   Serializable
    [SerializeField] protected AudioClip death;   Serializable

    [SerializeField] protected ParticleSystem mainEngineParticles;
    [SerializeField] protected ParticleSystem successParticles;
    [SerializeField] protected ParticleSystem deathParticles;   Se

    protected Rigidbody rigidBody;
    protected AudioSource audioSource;

    protected bool isTransitioning = false;
    protected bool collisionsDisabled = false;
    protected bool isSandboxMode = false;
```

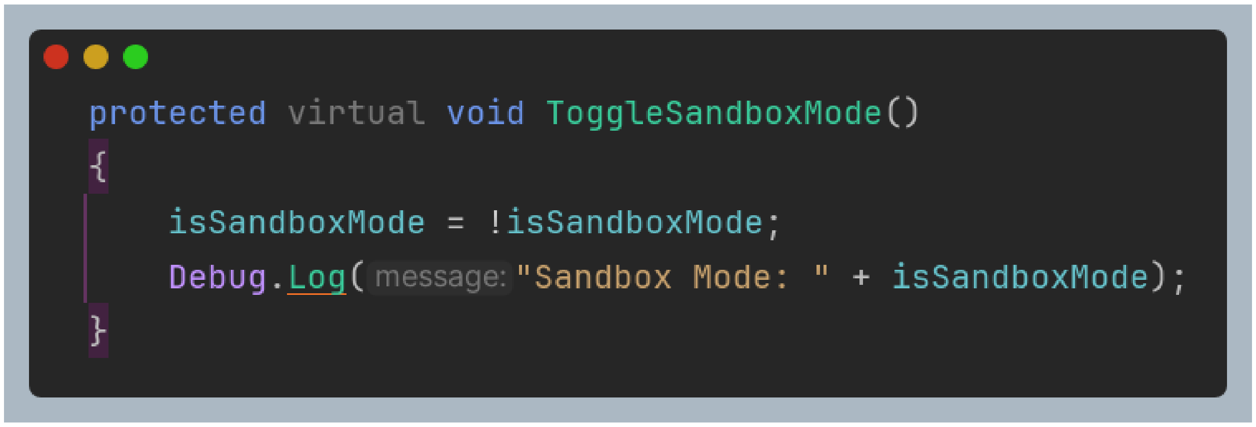
One feature in the game where this pattern is utilized is the ability to toggle sandbox mode. In the Spacecraft class, there's a method named ToggleSandboxMode(). This method toggles the isSandboxMode variable.



```
public class Rocket : Spacecraft
{

    Frequently called   0+1 usages   Haroon Sohail
    protected override void RespondToThrustInput()
    {
        if (isSandboxMode || Input.GetKey(KeyCode.Space))
        {
            ApplyThrust();
        }
        else
        {
            StopApplyingThrust();
        }
    }

    Frequently called   0+1 usages   Haroon Sohail
    protected override void RespondToRotateInput()
    {
        if (isSandboxMode || Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.D))
        {
            if (Input.GetKey(KeyCode.A))
            {
                RotateManually( rotationThisFrame:rcsThrust * Time.deltaTime);
            }
            else if (Input.GetKey(KeyCode.D))
            {
                RotateManually( rotationThisFrame:-rcsThrust * Time.deltaTime);
            }
        }
    }
```

In sandbox mode, collision responses are disabled (collisionsDisabled flag is set to true), allowing the spacecraft to pass through objects without triggering success or death sequences. This feature provides a sandbox environment where players can freely explore without the constraints of the main game's rules.


```
protected bool collisionsDisabled = false;
protected bool isSandboxMode = false;
```


```
protected virtual void ToggleSandboxMode()
{
    isSandboxMode = !isSandboxMode;
    Debug.Log( message: "Sandbox Mode: " + isSandboxMode);
}
```

Advantages:

Flexibility: The Subclass Sandbox pattern allows for flexibility in extending and customizing behavior. Subclasses can override specific methods to provide unique functionality while still leveraging the base class's template.

Code Reusability: By defining common functionality in the base class, the pattern promotes code reuse. Subclasses inherit common behavior and can focus on implementing or extending specific features.

Encapsulation: The base class encapsulates shared functionality, providing a clear separation of concerns. Subclasses only need to focus on implementing their specific behavior without worrying about the overall structure.

Disadvantages:

Complexity: Managing multiple subclasses and their interactions with the base class can lead to increased complexity, especially as the system grows larger.

Potential Tight Coupling: Subclasses may become tightly coupled with the base class, making it harder to modify or extend behavior without affecting other parts of the system.

Limited Extensibility: While subclasses can provide customization, the overall structure defined by the base class may limit extensibility in certain cases, especially if new requirements emerge that are not well-suited to the existing structure.

Alternative Design Patterns:

Decorator Pattern: Instead of using subclasses to customize behavior, the Decorator pattern could be used to dynamically add or modify behavior at runtime. Each decorator class could encapsulate a specific modification, allowing for more flexible composition of behaviors.

Strategy Pattern: This pattern could be used to define a family of algorithms (strategies) for handling different modes or behaviors. Each strategy would encapsulate a specific behavior, and the spacecraft class could be configured with the appropriate strategy based on the current mode.

State Pattern: Instead of toggling sandbox mode with a boolean flag, the State pattern could be used to represent different states of the spacecraft (e.g., normal mode, sandbox mode). Transitions between states could trigger changes in behavior, allowing for more dynamic control over functionality.

Why Subclass Sandbox Pattern was Chosen:

In this context, the Subclass Sandbox pattern was chosen because it provides a simple and straightforward way to extend and customize behavior for different modes (normal mode and sandbox mode). It allows for clear separation of concerns and promotes code reuse while offering enough flexibility to accommodate different gameplay requirements. Additionally, the simplicity of the feature and the clear distinction between modes make the Subclass Sandbox pattern a suitable choice for this implementation.