State Machine Pattern:

I'm sure you've worked on making the player move before. Here's what we'd typically do: attach a Character Controller or Rigidbody, create a PlayerMovement class, and put all the movement functions inside it like handling idle, walking, jumping, swimming, etc. This works fine at first. But as the project grows and you add more player mechanics like making the player limp when hurt, preventing swimming when health is low, adding collision detection, and so on the class can get pretty messy. With a bunch of nested if-else if statements, it becomes harder to manage, and the chances of bugs and errors increase in a jumbled-up class like this.

```
public class PlayerMovement : MonoBehaviour
{
    public float movementSpeedLimp;
    public float movementSpeed;

    private InputAction moveAction;
    private InputAction attackAction;

    public Animator animator;

    public Rigidbody Rigidbody;

    private bool isAttacking = false;
    private bool isHurt = false;

    Unity Message | 0 references
    private void Start()
    {
        moveAction = InputSystem.actions.FindAction("Move");
        attackAction = InputSystem.actions.FindAction("Attack");
    }

    Unity Message | 0 references
    private void Update()
    {
        Vector2 moveValue = moveAction.ReadValue<Vector2>();
        bool attackValue = attackAction.IsPressed();

        if (attackValue && !isAttacking)
        {
            if (isHurt)
            {
                // attack slow!
                animator.SetTrigger("AttackSlow");
            }
            else
            {
                // attack!
                animator.SetTrigger("AttackSlow");
            }

            isAttacking = true;
        }
        else if (!isAttacking && moveValue.magnitude > 0)
        {
            if (isHurt)
            {
                // walk slow!
                Rigidbody.MovePosition(moveValue * movementSpeedLimp * Time.deltaTime);
                animator.SetTrigger("WalkLimp");
            }
            else
            {
                Rigidbody.MovePosition(moveValue * movementSpeed * Time.deltaTime);
                animator.SetTrigger("Walk");
            }
            isAttacking = false;
        }
        else
        {
            isAttacking = false;
            animator.SetTrigger("Idle");
        }
    }
}
```

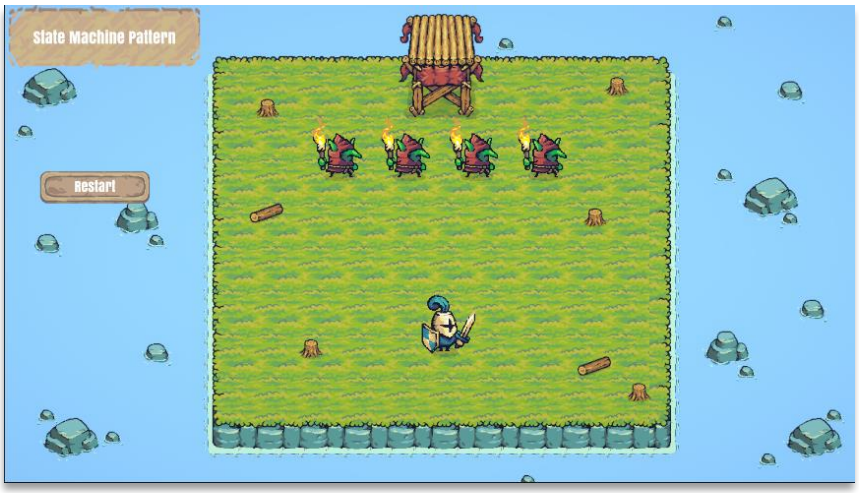
A great solution to this problem is to break the actions into different states, like Idle State, Walk State, Attack State, etc. Each of these states would handle its own specific behavior and operations. This makes the code much more readable, manageable, scalable, and loosely coupled.

The State Machine Pattern can be used in various situations, not just for player movement. If you've worked with Unity's animation system, you've already seen this pattern in action because Unity's animation system is built on a state machine. You can apply it to a season system, for example, where in the Winter State, the land has snow, and in the Summer State, the land is dry. You can even use it with audio, like playing specific sounds based on the state. The possibilities depend on your project needs.

In Unity C#, you can create a custom state machine system by:

- Making an abstract state class with functions like EnterState(), UpdateState(), and any other necessary methods that all states will use as a foundation.
- Creating specific state classes that inherit from the abstract state class and implementing the unique behavior for each state.
- Building a concrete class to manage state creation, switching between states, and handling state-specific functions.

To make it more clear, imagine a Knight vs Goblin setup. We want the Knight to be able to walk, attack, or be in idle when doing nothing, using the State Machine Pattern. Also, when the Knight's sword strikes the Goblin, the Goblin should die.



First, we create an Abstract BaseState class having the abstract functions

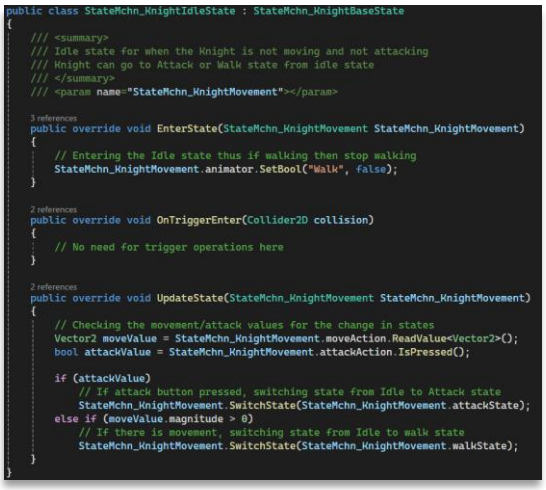
```
public abstract class StateMchn_KnightBaseState
{
    /// <summary>
    /// This is an abstract class responsible for having the necessary functions that all the states will have
    /// </summary>
    /// <param name="StateMchn_KnightMovement"></param>

    // For when the state is selected, this function is a must
    5 references
    public abstract void EnterState(StateMchn_KnightMovement StateMchn_KnightMovement);

    // For each update action performed by the state per frame, this function is a must
    4 references
    public abstract void UpdateState(StateMchn_KnightMovement StateMchn_KnightMovement);

    // Trigger detection
    4 references
    public abstract void OnTriggerEnter(Collider2D collision);
}
```

Next, we create Idle, Walk, Attack state classes respectively implementing the BaseState abstract class. And having the necessary code in it according to the State.



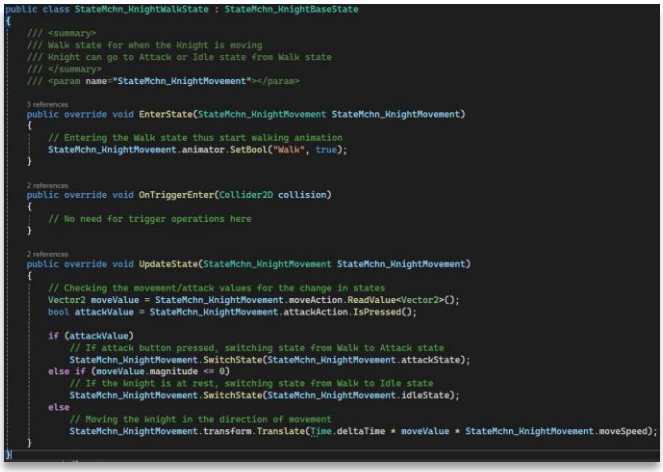
```
public class StateMchn_KnightIdleState : StateMchn_KnightBaseState
{
    /// <summary>
    /// Idle state for when the Knight is not moving and not attacking
    /// Knight can go to Attack or Walk state from idle state
    /// </summary>
    /// <param name="StateMchn_KnightMovement"></param>

    3 references
    public override void EnterState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Entering the Idle state thus if walking then stop walking
        StateMchn_KnightMovement.animator.SetBool("Walk", false);
    }

    2 references
    public override void OnTriggerEnter(Collider2D collision)
    {
        // No need for trigger operations here
    }

    2 references
    public override void UpdateState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Checking the movement/attack values for the change in states
        Vector2 moveValue = StateMchn_KnightMovement.moveAction.ReadValue<Vector2>();
        bool attackValue = StateMchn_KnightMovement.attackAction.IsPressed();

        if (attackValue)
            // If attack button pressed, switching state from Idle to Attack state
            StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.attackState);
        else if (moveValue.magnitude > 0)
            // If there is movement, switching state from Idle to walk state
            StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.walkState);
    }
}
```



```
public class StateMchn_KnightWalkState : StateMchn_KnightBaseState
{
    /// <summary>
    /// Walk state for when the Knight is moving
    /// Knight can go to Attack or Idle state from Walk state
    /// </summary>
    /// <param name="StateMchn_KnightMovement"></param>

    3 references
    public override void EnterState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Entering the Walk state thus start walking animation
        StateMchn_KnightMovement.animator.SetBool("Walk", true);
    }

    2 references
    public override void OnTriggerEnter(Collider2D collision)
    {
        // No need for trigger operations here
    }

    2 references
    public override void UpdateState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Checking the movement/attack values for the change in states
        Vector2 moveValue = StateMchn_KnightMovement.moveAction.ReadValue<Vector2>();
        bool attackValue = StateMchn_KnightMovement.attackAction.IsPressed();

        if (attackValue)
            // If attack button pressed, switching state from Walk to Attack state
            StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.attackState);
        else if (moveValue.magnitude <= 0)
            // If the knight is at rest, switching state from Walk to Idle state
            StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.idleState);
        else
            // Moving the knight in the direction of movement
            StateMchn_KnightMovement.transform.Translate(Time.deltaTime * moveValue * StateMchn_KnightMovement.moveSpeed);
    }
}
```

```
public class StateMchn_KnightAttackState : StateMchn_KnightBaseState
{
    /// <summary>
    /// Attack state for when the Knight is attacking
    /// Knight can go to Idle or Walk state from Attack state
    /// </summary>

    private float _timer = 0f;

    private StateMchn_Goblin _StateMchn_Goblin;

    3 references
    public override void EnterState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Entering the Attack state thus start attack animation
        StateMchn_KnightMovement.animator.SetTrigger("Attack");
    }

    2 references
    public override void OnTriggerEnter(Collider2D collision)
    {
        // Detecting if the goblin in in trigger range of the sword
        if (collision.CompareTag("Goblin"))
            // caching the attacked goblin for later use
            _StateMchn_Goblin = collision.GetComponent<StateMchn_Goblin>();
    }

    2 references
    public override void UpdateState(StateMchn_KnightMovement StateMchn_KnightMovement)
    {
        // Checking when the sword animation will end
        if (_timer > StateMchn_KnightMovement.attackClip.length)
        {
            // if goblin attacked, making the cached goblin die
            _StateMchn_Goblin?.Die();

            _timer = 0f;

            // Checking the movement values for the change in states
            if (StateMchn_KnightMovement.moveAction.ReadValue<Vector2>().magnitude > 0)
                // If moving then switch from Attack to Walk State
                StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.walkState);
            else
                // If knight is at rest, switch from Attack to Idle State
                StateMchn_KnightMovement.SwitchState(StateMchn_KnightMovement.idleState);
        }
        else
            // Increasing sword animation end wait timer by delta time per frame
            _timer += Time.deltaTime;
    }
}
```

We need to have a concreate KnightMovement class for creating, handling and switching the states. By default the knight will be at idle state. Update will call the current state's update to perform operation according to the state the Knight is in.

```
public class StateMchn_KnightMovement : MonoBehaviour
{
    /// <summary>
    /// Class responsible for knight's movement
    /// Handling the states as required
    /// </summary>

    public float moveSpeed; // Knight movement speed

    // Knight's different movement states
    internal StateMchn_KnightIdleState idleState = new StateMchn_KnightIdleState();
    internal StateMchn_KnightWalkState walkState = new StateMchn_KnightWalkState();
    internal StateMchn_KnightAttackState attackState = new StateMchn_KnightAttackState();

    private StateMchn_KnightBaseState _currentState;

    // Input for Knight's movement and attack
    internal InputAction moveAction;
    internal InputAction attackAction;

    internal Animator animator;

    public AnimationClip attackClip;

    Unity Message | 0 references
    private void Start()
    {
        animator = GetComponent<Animator>();

        moveAction = InputSystem.actions.FindAction("Move");
        attackAction = InputSystem.actions.FindAction("Attack");

        // By default initially Knight will be in the Idle State
        _currentState = idleState; // Setting the current state as Idle State
        _currentState.EnterState(this); // Entering the state for the state to take charge and perform relevent operations
    }

    Unity Message | 0 references
    private void Update()
    {
        _currentState.UpdateState(this); // Updates per frame, performed by the current state we are in
    }

    6 references
    public void SwitchState(StateMchn_KnightBaseState StateMchn_KnightBaseState)
    {
        // This method switches the current state to the provided state
        _currentState = StateMchn_KnightBaseState;
        StateMchn_KnightBaseState.EnterState(this);
    }

    Unity Message | 0 references
    private void OnTriggerStay2D(Collider2D collision)
    {
        // For trigger detection
        _currentState.OnTriggerEnter(collision);
    }
}
```

Now, go and check the scripts and play the project and try out yourself!