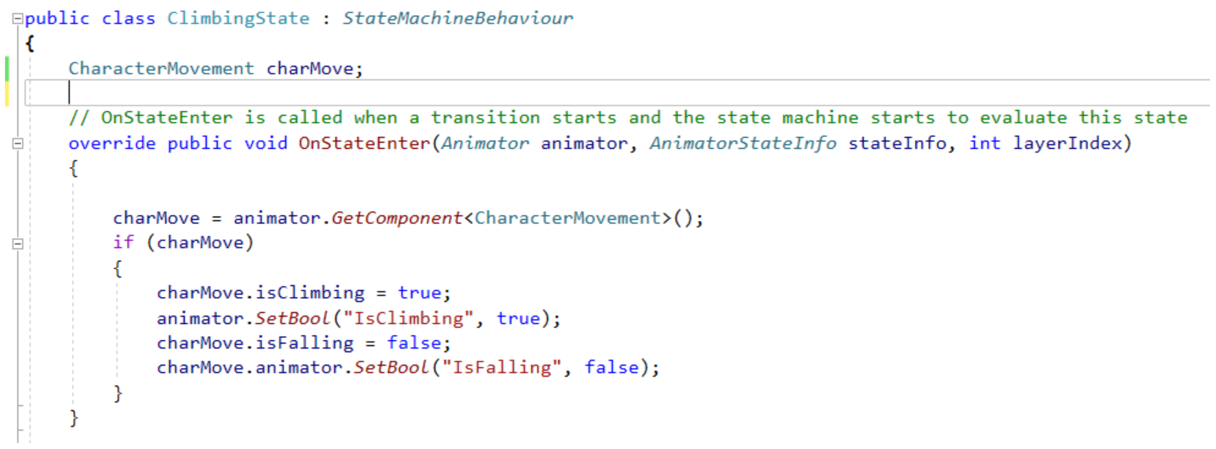
README.code
Prototype Four
Gabriel Campell

There are two player characters (and you swap between them with the ‘P’ button). The first one is a somewhat complex controller using more of Unity’s built in State pattern behavior scripts (StateMachineBehaviour) on individual Animator window states, and the CharacterMovement script to drive most things (located directly under Assets). CharacterMovement is pretty straightforward and just gets horizontal and vertical axis input, and left shift input, and feeds that into some walk and rotate speed math, and that information is used to set speed floats, and walking states for the animator (taking into account that you can go a negative direction, and that states should be blended a bit):

```
float translation = Input.GetAxis("Vertical") * speed * Time.deltaTime;
float rotation = Input.GetAxis("Horizontal") * rotationSpeed * Time.deltaTime;
animator.SetFloat("Rotation", Input.GetAxis("Horizontal"));

if (Input.GetKey(KeyCode.LeftShift))
{
    currentSpeed += translation;
    maxSpeed = runSpeed;
    if (currentSpeed > maxSpeed)
        currentSpeed = maxSpeed;
    if (currentSpeed < minSpeed)
        currentSpeed = minSpeed;
}
else
{
    maxSpeed = walkSpeed;
    if (currentSpeed > maxSpeed)
        currentSpeed -= Time.deltaTime * speed;
    else if (currentSpeed < minSpeed)
        currentSpeed = minSpeed;
    else
        currentSpeed += translation;
}
```

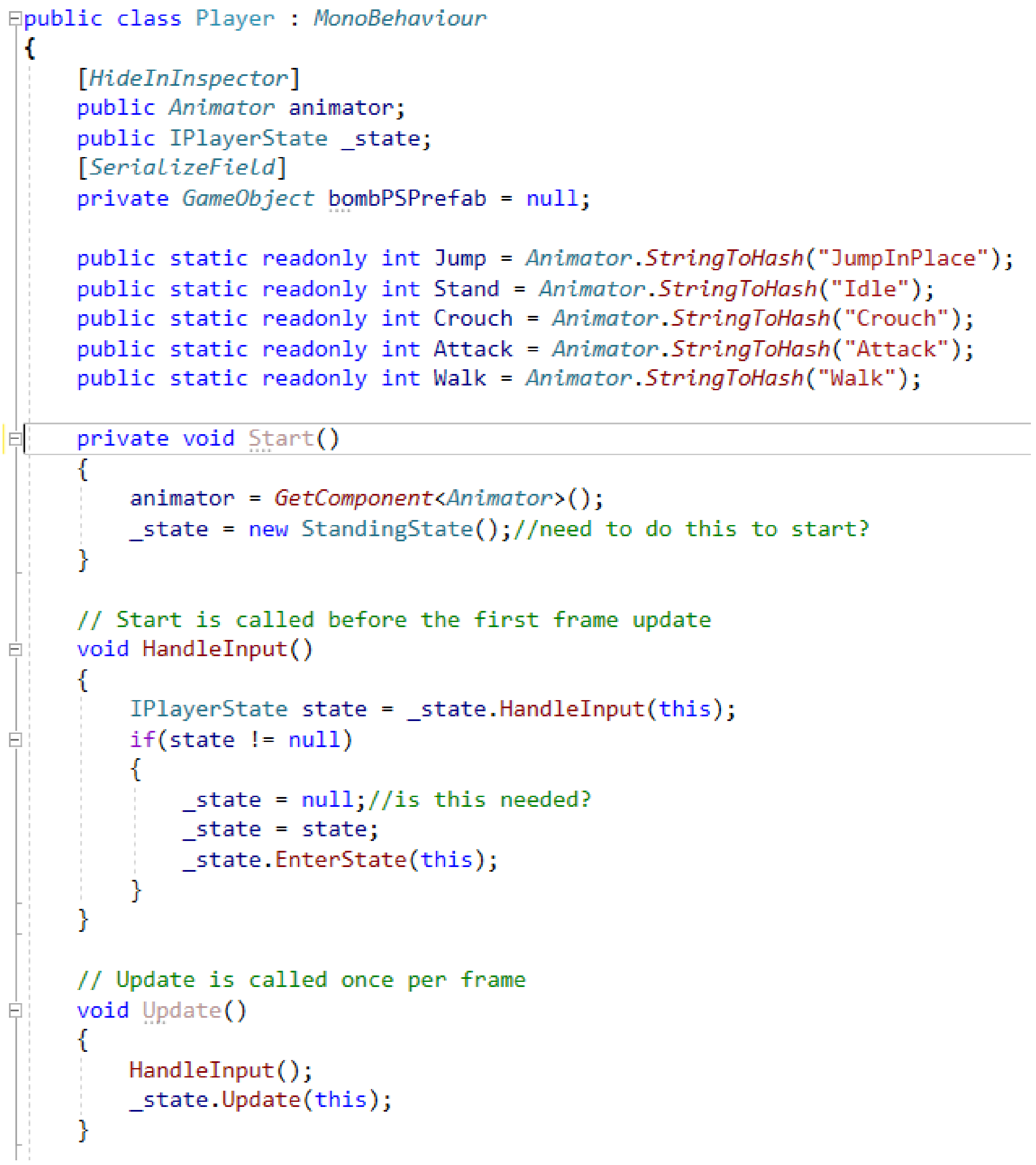
It does some similar things for the climbing directions as well. I have a few scripts hooked up to trigger colliders that set some bools and animator parameters for when you jump up onto the wall, get to the top of it, or go back down to step off of it (ClimbStartScript, ClimbGroundScript and ClimbEndScript, all of which are also just in Assets directly). I also make use of quite a few StateMachineBehaviour scripts, attached to animation states in the Animator window, for when I need to set similar booleans. For example, when entering ClimbingState.cs:



```
public class ClimbingState : StateMachineBehaviour
{
    CharacterMovement charMove;
    
    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {

        charMove = animator.GetComponent<CharacterMovement>();
        if (charMove)
        {
            charMove.isClimbing = true;
            animator.SetBool("IsClimbing", true);
            charMove.isFalling = false;
            charMove.animator.SetBool("IsFalling", false);
        }
    }
```

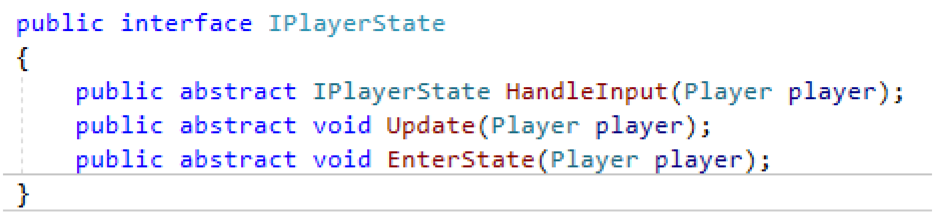
I have similar StateMachineBehaviour scripts in the form of IdleState.cs, RollBehavior.cs, ClimbToTopState.cs, JumpState.cs, and DropHangState.cs, all in the Assets folder as well.

In addition to using the existing Unity StateMachineBehaviour scripts for a more complex controller, I also thought I should implement the State pattern myself to learn the material a bit better, so I added IPlayerState.cs, a file with the IPlayerState and state classes for Standing, Crouching, Jumping, Walking and Attacking that all implement it, via a Player.cs class (these are in Assets/StatePatternScripts). My Player class is similar to the book’s Heroine class, but in Unity/C#, where HandleInput will swap the state if the input called into the current _state returns a new state, then it calls the EnterState method on that new state. Update is just always watching for if HandleInput actually Input anything (I don’t pass in input like in the book as Unity’s Input system is kind of built, it seems, just to be called directly rather than be passed in as a parameter?):

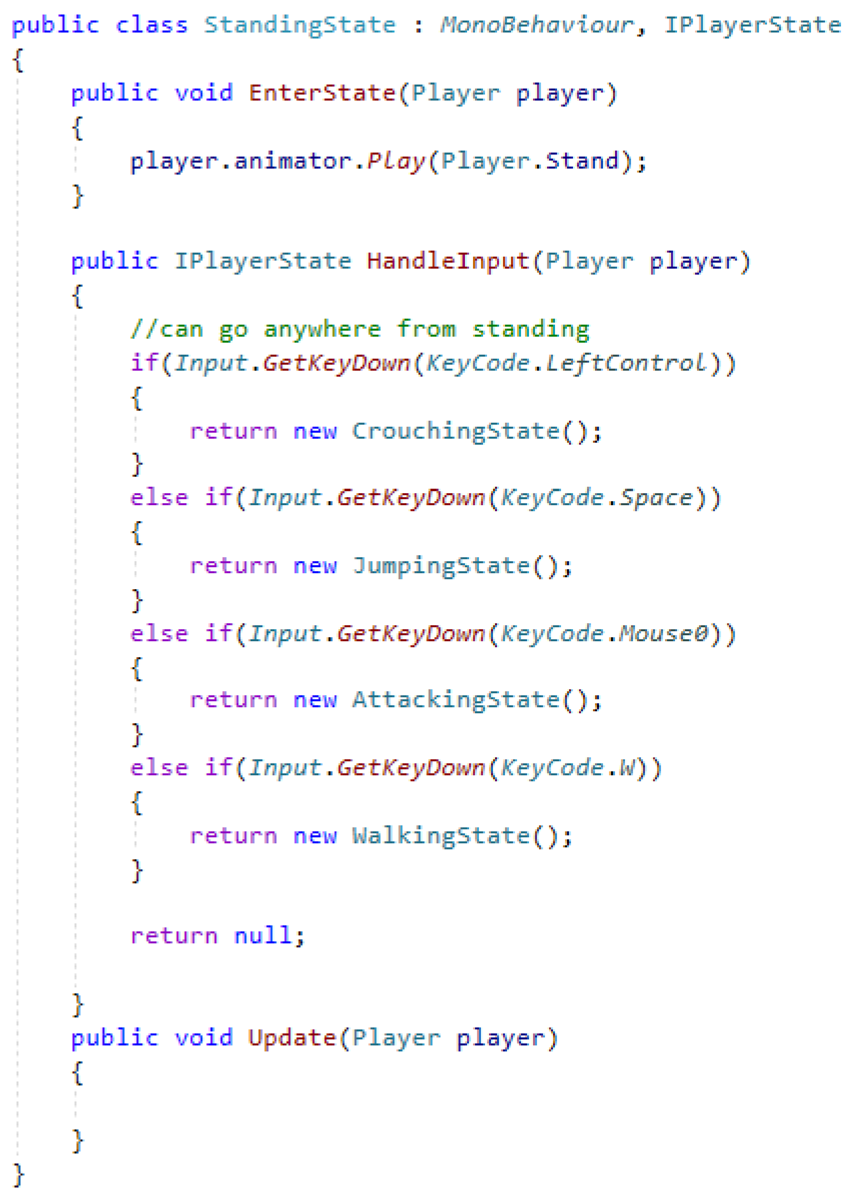


```
public class Player : MonoBehaviour
{
    [HideInInspector]
    public Animator animator;
    public IPlayerState _state;
    [SerializeField]
    private GameObject bombPSPrefab = null;

    public static readonly int Jump = Animator.StringToHash("JumpInPlace");
    public static readonly int Stand = Animator.StringToHash("Idle");
    public static readonly int Crouch = Animator.StringToHash("Crouch");
    public static readonly int Attack = Animator.StringToHash("Attack");
    public static readonly int Walk = Animator.StringToHash("Walk");

    private void Start()
    {
        animator = GetComponent<Animator>();
        _state = new StandingState();//need to do this to start?
    }

    // Start is called before the first frame update
    void HandleInput()
    {
        IPlayerState state = _state.HandleInput(this);
        if(state != null)
        {
            _state = null;//is this needed?
            _state = state;
            _state.EnterState(this);
        }
    }

    // Update is called once per frame
    void Update()
    {
        HandleInput();
        _state.Update(this);
    }
```

And here's my interface and an example state showing how it returns a new state of a given kind based on input, and without input it will just return null and thus not trigger a new state to enter:



```
public interface IPlayerState
{
    public abstract IPlayerState HandleInput(Player player);
    public abstract void Update(Player player);
    public abstract void EnterState(Player player);
}
```



```
public class StandingState : MonoBehaviour, IPlayerState
{
    public void EnterState(Player player)
    {
        player.animator.Play(Player.Stand);
    }

    public IPlayerState HandleInput(Player player)
    {
        //can go anywhere from standing
        if(Input.GetKeyDown(KeyCode.LeftControl))
        {
            return new CrouchingState();
        }
        else if(Input.GetKeyDown(KeyCode.Space))
        {
            return new JumpingState();
        }
        else if(Input.GetKeyDown(KeyCode.Mouse0))
        {
            return new AttackingState();
        }
        else if(Input.GetKeyDown(KeyCode.W))
        {
            return new WalkingState();
        }

        return null;

    }
    public void Update(Player player)
    {

    }
}
```

You can also see that my EnterState method calls the animator to just directly play the proper animation without relying on any Animator window Parameters like triggers.

I think that the understandability of my simpler state machine is much better than than my first, more complex one using Unity's StateMachineBehaviour, as the latter is split across many scripts and is trying to juggle a lot of booleans and other states like isKinematic off and on, as well as root motion, animation blends, transition interrupts, and moving the character along a curve as the animation plays, so there's a lot going on that makes the code difficult to follow. I think I need to re-analyze my IsClimbing/Jumping/Soaring/Walking bools and states to make them transition more consistently. I'd be interested in seeing whatever Unity Engine Industry standards are for this sort of thing, because although I'm using the Unity built-in State pattern, I feel like I'm running into a lot of the issues that motivate using the State pattern in the first place where I have these *ad hoc* booleans controlling state rather than encapsulated states only knowing what they need to (e.g. IsClimbing gets set all over the place). Maybe I could have done more playing animations directly in the more complex animator without relying on the transitions and complex blends and interruptions. I also rand into a lot of complex issues with root motion, as I used that to drive the actual movement (I don't call out to translate), but sometimes that gets a bit messy, and there are a lot of settings like original vs center of mass, looping, XZ/Y root and offsets and so on. I like the simplicity and consistency of the version I implemented myself, but it is also much less powerful. A more complex custom implementation of concurrent or hierarchical state machines might help out with doing some of the things from Unity's animator layers and additive/masking functionality.