Prototype Two Readme.code
Gabriel Campbell

I used the Command Pattern to track moves and pickups by the player, and just moves by enemy AI in order to be able to push and pop them onto stacks and thus record them for Undo/Redo purposes. I also specifically tracked the direction of movement rather than the previous (x,y) position, so that each move only needs to know what direction to reverse to undo (I just made an enum MovementDirections for directions, and utility functions to reverse directions and convert them to transform changes).

All Scripts are in Assets\Scripts\. My implementation was largely drawn from the Nystrom book, but adapted for Unity/C#:

```
public abstract class Command
{
    //don't need a destructor as c#
    public abstract bool Execute();
    public abstract bool Undo();

}
```

A child MoveUnitCommand class, used by both the AI and Player Units:

```
class MoveUnitCommand : Command
{
    Unit unit;
    MovementDirections movementDirection;
    MovementDirections reverseDirection;

    public MoveUnitCommand(ref Unit _unit, MovementDirections _movementDirection)
    {
        unit = _unit;
        reverseDirection = GetOppositeDirection(_movementDirection);
        movementDirection = _movementDirection;

    }
```

The Execute() and Undo() overrides just check if a Unit.cs CanMove() and then call Move() on the UnitMovement.cs component of the Unit in question, ie the movement code itself is modular and a separate piece from the Command code. PickupItemCommand.cs, also a child of Command.cs, simply tracks a ref to a Unit and a Pickup.cs class object (of which there is only one child currently, Coin.cs):

```
public class PickupItemCommand : Command
{
    Unit unit;
    Pickup pickup;

    public PickupItemCommand(ref Unit _unit, Pickup _pickup)
    {
        unit = _unit;
        pickup = _pickup;
    }

    public override bool Execute()
    {
        if(unit.unitInventory.AddPickup(pickup))
        {
            pickup.TurnOff();
            return true;
        }
        return false;
    }

    public override bool Undo()
    {
        if(unit.unitInventory.RemovePickup(pickup))
        {
            pickup.TurnOn();
            return true;
        }
        return false;
    }
}
```

I like to return Booleans for these Execute() and Undo() functions, so that a check can be easily made to ensure that they can only be added on to Stack<Command> stacks if they can actually be done or undone.

PlayerInput.cs takes horizontal and vertical axis input, converts it into a MoveDirections enum, and uses that and the Player.cs Unit ref to make a new Command inside of HandleInput, which is basically the C# version of the C++ HandleInput from the Nystrom text. It is used like so:

```
void Update()
{
    timeSinceLastMove += Time.deltaTime;
    bool choseMove = true;
    moveCommand = (MoveUnitCommand)HandleInput(ref choseMove);
    if((timeSinceLastMove > timeBetweenMoves) && choseMove)
    {
        if(moveCommand.Execute())
        {
            timeSinceLastMove = 0;
            OnMoved();
            thisUnit.commands.Push(moveCommand);
            thisUnit.redoCommands.Clear();
        }
    }
```

All Unit class objects, Player and Enemy being the derived classes, have on that base class their commands and redoCommands stacks for tracking movements. When the player makes a move in PlayerInput, it (using a version of the Observer pattern in the form of C# delegates) broadcasts that move, to which all AIInput.cs class objects are subscribed, telling them to move as well. Double Buffer pattern would have helped to make move orders more consistent if I were to add attacks or something later.

Unity's tile/tilemaps are likely using Flyweight already, as a single sprite reference may be used across an infinite number of tile instances, and similarly the C# Delegates I used are a variation on the Observer pattern, so I didn't need to implement Observer or Flyweight, but I did end up relying on them. I could have used the Prototype Pattern for Unit.cs base Enemy/Player classes, or Pickup.cs base Coin variations. This would have perhaps enabled runtime spawning of a large variety of different enemies and coins from a single spawner for each type, but the game design does not rely on runtime spawning or a wide variety of Unit or Pickup types, and Unity's existing prefab system was adequate for any instantiation needs (including TilePrefabs for the Coins). If I were to use a Prototype Pattern spawner system to clone enemies, would I also clone their Stack<Command> of commands for undo and redo purposes? Because I only track command *directions* and not *locations*, there would not be a location state to clone, so would I perhaps instantiate the new Enemy Unit at runtime in a random location, but with the same memory of undoable direction movements, such that undoing moves would create the same pattern, but offset by the original distance and direction between itself and its progenitor.

I learned some new patterns and techniques, both for Command/Undo/Redo, and in using C# refs, that I did not know about before. It's great to be able to use/reuse code that passes refs to itself in the constructor and thus knows to record or alter its own state. One issue I noticed is that the more complex the game becomes, and especially the more moves and state changes are made simultaneously, the more onerous it becomes to keep up with recording everything and ensuring that undos/redos are correct, and don't run into issues like being able to infinitely pickup a coin. Because most of my Commands are filtered through the PlayerInput at some point, player moves and undos balloon in complexity as the game gets larger. I might do it differently with some kind of CommandsManager, rather than just having stacks of Commands on Units.