Command Pattern:

The Command pattern can be used in many ways, depending on the situation thus can have multiple definitions. Basically, as the name suggests, it's about storing, sequencing, and executing custom commands/actions/events whenever you need. It can also be used for actions like undo/redo, or storing player input bindings. For example, in tactical games, a player might click to perform an action at different locations, and the character follows those commands in sequence. It can also be used for turn-based games like Pokémon battles or many other cases, really it all depends on your project's needs.

Typically, the Command pattern involves an interface or an abstract class with an Execute() method. Any class that implements or inherits from this can define a custom command with its own functionality. You also need an invoker class that handles adding, removing, and processing commands. The concrete class would tell the invoker to add and execute the custom command as needed.

Let’s imagine a scenario with a Knight vs. Goblin setup. The King Goblin has 4 goblins as his troop. He commands his goblin soldiers to march towards the knight one by one, and the knight kills them when they come into contact.

To implement this, we create an abstract class for the command and a custom command class that inherits from it. We also have an IsFinished method to check if the current command has finished executing only then will we move on to the next one. These classes are non-MonoBehaviour, and I'm using a constructor to specify which enemy was sent by the King Goblin.

```
public abstract class Cmnd_ICommand
{
    // interface for command that will be implemented in custom command classes
    2 references
    public abstract void Execute(); // must have this method

    public bool isFinished = false; // to check if the current command has finished executing
}
```

```
public class Cmnd_SendEnemyCommand : Cmnd_ICommand
{
    /// <summary>
    /// Custom sendenemy command
    /// King goblin will use this to send the required enemy towards the knight
    /// </summary>

    private Cmnd_GoblinTroop _Cmnd_GoblinTroop;

    1 reference
    public Cmnd_SendEnemyCommand(Cmnd_GoblinTroop Cmnd_GoblinTroop)
    {
        // caching the goblin that the king goblin has commanded to march, but waiting to be executed
        _Cmnd_GoblinTroop = Cmnd_GoblinTroop;
    }

    2 references
    public override void Execute()
    {
        // it's this goblins turn to march toward knight, execting
        _Cmnd_GoblinTroop.StartMarch(this);
    }
}
```

We also need a command invoker class, which is responsible for adding and processing commands in our case. Since we're not dealing with undo/redo features here, the invoker's job is pretty straightforward, just handling the sequence of commands.

I've used a Queue in this example because I want the goblins to march in a First-In-First-Out (FIFO) sequence. Of course, you can choose a different data structure (like a Stack, List, or Dictionary) based on your specific needs. The key point is that the choice of structure depends on how you want to organize the order of the commands.

Additionally, I'm ensuring that only one command is executed at a time. In other words, a goblin won't start marching until the current one has been "finished" (for example, when the goblin dies). So, once one goblin is done, the next one in the queue can begin its march. This ensures a strict sequence where each action waits for the previous one to finish.

```
public class Cmnd_SendEnemyInvoker : MonoBehaviour
{
    /// <summary>
    /// Invoker of commands
    /// Stores the commands into a data structure and executes the action as required
    /// </summary>

    // commands queue, to store command and execute as First in First Out (FIFO)
    private Queue<Cmnd_ICommand> commandsQueue = new Queue<Cmnd_ICommand>();
    private Cmnd_ICommand _currentCommand;

    // Array of goblin troops which can be commanded to march
    [SerializeField] private Cmnd_GoblinTroop[] Cmnd_GoblinTroopsArr;

    1 reference
    public void AddCommand(int member)
    {
        // Add send enemy command to the queue, with the king goblin's desired enemy to send
        Cmnd_SendEnemyCommand command = new Cmnd_SendEnemyCommand(Cmnd_GoblinTroopsArr[member - 1]);
        commandsQueue.Enqueue(command);
    }

    Unity Message | 0 references
    private void Update()
    {
        // Checking every frame if there are commands in the queue and next commmand can be processed
        ProcessCommand();
    }

    1 reference
    private void ProcessCommand()
    {
        // if invoker has a current command in hand
        // and is not finished then wait for it to complete before going to the next one
        if (_currentCommand != null && !_currentCommand.isFinished)
            return;

        // if there are no other commands remaining in the queue, do nothing
        if (!commandsQueue.Any())
            return;

        // execute the next in queue command
        _currentCommand = commandsQueue.Dequeue();
        _currentCommand.Execute();
    }
}
```

We also have a concrete King Goblin class, which is responsible for adding commands to the invoker for execution. In our case, the King Goblin gives the command for a specific goblin to march towards the knight. The command specifying which goblin should go is sent to the invoker, and the invoker processes it accordingly.

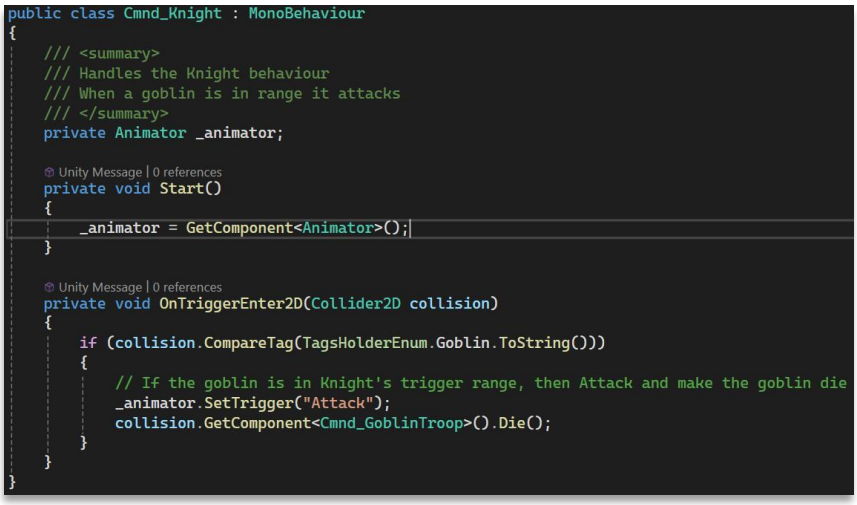
So, whenever the King Goblin wants a goblin to march, it creates the appropriate command (for a specific goblin) and adds it to the invoker's queue. The invoker will then handle executing these commands in order, making sure that each goblin marches toward the knight one after another.

```
public class Cmnd_KingGoblin : MonoBehaviour
{
    /// <summary>
    /// King Goblin, responsible for command the enemy goblins towards the knight
    /// </summary>
    ///
    public GameObject cmndSpriteGObj;

    [SerializeField] private Animator animator;

    private int _sendEnemyCount = 0;

    private Cmnd_SendEnemyInvoker _Cmnd_SendEnemyInvoker; // The invoker of the commands, where commands would be sent for execution

    // Start is called once before the first execution of Update after the MonoBehaviour is created
    Unity Message | 0 references
    private void Start()
    {
        _Cmnd_SendEnemyInvoker = GetComponent<Cmnd_SendEnemyInvoker>();
    }

    2 references
    internal void CommandEnemy()
    {
        // Checking which enemy to send, current in order 1-4, you can make it random too!
        if (_sendEnemyCount >= 4)
        {
            CmndSpriteActivation(false, "");
            return;
        }

        _sendEnemyCount++;

        AnimateTrigger("Go");

        // king goblin shows which enemy he has commanded
        string msg = "GO " + _sendEnemyCount + "!";
        CmndSpriteActivation(true, msg);

        // adding the command to the invorker for execution mechanism
        _Cmnd_SendEnemyInvoker.AddCommand(_sendEnemyCount);
    }

    1 reference
    private void AnimateTrigger(string trigger)
    {
        animator.SetTrigger(trigger);
    }

    2 references
    private void CmndSpriteActivation(bool toShow, string text)
    {
        cmndSpriteGObj.SetActive(toShow);
        cmndSpriteGObj.transform.GetComponentInChildren<TextMeshPro>().text = text;
    }
}
```

We also have a goblin class responsible for the marching and die mechanism, when the command execution is given by the invoker.

```
public class Cmnd_GoblinTroop : MonoBehaviour
{
    /// <summary>
    /// Responcible for Goblin to be commanded march and die mechanism
    /// </summary>

    [SerializeField] private Transform knightTransfrm;

    private Animator _animator;

    private bool _toMarch = false;

    public GameObject skull;

    private Cmnd_ICommand _Cmnd_ICommand;

    [SerializeField] private Cmnd_KingGoblin Cmnd_KingGoblin;

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

    Unity Message | 0 references
    private void Update()
    {
        March();
    }

    #region MarchMech
    1 reference
    private void March()
    {
        if (_toMarch)
        {
            // march towards the knight
            transform.position = Vector2.MoveTowards(transform.position, knightTransfrm.position, Time.deltaTime);

            if (Vector2.Distance(transform.position, knightTransfrm.position) <= 0.1f)
                _toMarch = false;
        }
    }

    1 reference
    internal void StartMarch(Cmnd_ICommand Cmnd_ICommand)
    {
        // Gotten the command from king goblin to march towards the knight!
        _Cmnd_ICommand = Cmnd_ICommand;
        _toMarch = true;
        _animator.SetTrigger("Walk");
    }
    #endregion MarchMech

    #region DieMech
    1 reference
    internal void Die()
    {
        Invoke(nameof(DieAfterDelay), 0.2f);
    }

    1 reference
    private void DieAfterDelay()
    {
        GetComponent<SpriteRenderer>().enabled = false;
        skull.SetActive(true);

        // finishing the current command execution
        _Cmnd_ICommand.isFinished = true;

        // letting the king goblin to add a new command if any
        Cmnd_KingGoblin.CommandEnemy();

        Invoke(nameof(HideTroop), 1f);
    }

    1 reference
    private void HideTroop()
    {
        gameObject.SetActive(false);
    }
    #endregion DieMech
}
```

We also have a knight class to detect if the goblin is in trigger range, attack and make it die!



```
public class Cmnd_Knight : MonoBehaviour
{
    /// <summary>
    /// Handles the Knight behaviour
    /// When a goblin is in range it attacks
    /// </summary>
    private Animator _animator;

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

    Unity Message | 0 references
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag(TagsHolderEnum.Goblin.ToString()))
        {
            // If the goblin is in Knight's trigger range, then Attack and make the goblin die
            _animator.SetTrigger("Attack");
            collision.GetComponent<Cmnd_GoblinTroop>().Die();
        }
    }
}
```

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