Historia de Usuario: Movimiento lateral del player separado del Input	
Identificador: UH04	Usuario: Programador
Prioridad en negocio: -	Riesgo en desarrollo: -
Puntos estimados: -	Iteración asignada: -
Programador Responsable: Ariel Vega	

Descripción

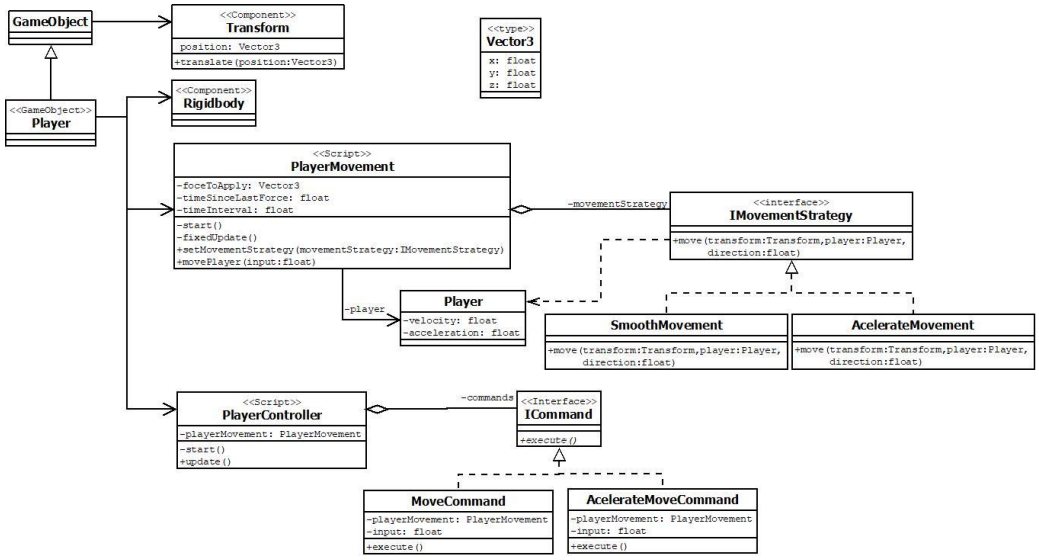
Como programador quiero que el movimiento del jugador esté separado del mecanismo de detección de comandos para facilitar el diseño del juego en diferentes plataformas.

Observaciones

Debe haberse implementado previamente la UH3. Si observa la implementación de los métodos Move() de cada tipo de movimiento, puede notar que se hace la lectura del Input en este método. Se debe separar el input de la ejecución del movimiento. Esto facilita el cambio de comandos, o el cambio del tipo de comandos (por ejemplo, en lugar de usar teclas se podría usar el touchpad, o el puntero del mouse, o la pantalla del celular).

Para este caso, se usará las teclas A y S, así como las flechas izquierda y derecha para el movimiento lateral, mientras que usaremos la barra espaciadora para dotar al mismo movimiento lateral una aceleración.

Diagrama de clases



Criterios de aceptación

- Se separa la detección del input de la ejecución del comando
- Cada comando está encapsulado en una nueva clase que invoca el movimiento

Actividad 01: Aplicar el patrón de diseño Command para separar el input del movimiento. Este patrón permite solicitar una operación a un objeto sin conocer realmente el contenido de esta operación, ni el receptor real de la misma. Para ello se encapsula la petición como un objeto, con lo que además facilita la parametrización de los métodos A continuación, describiremos el siguiente patrón de diseño.

Patrón de Diseño Command

Propósito:

- Encapsula un mensaje como un objeto, con lo que permite gestionar colas o registro de mensaje y deshacer operaciones.
- Soportar restaurar el estado a partir de un momento dado.
- Ofrecer una interfaz común que permita invocar las acciones de forma uniforme y extender el sistema con nuevas acciones de forma más sencilla

Motivación:

Si debe implementar un sistema para detectar las órdenes, comandos u ordenes que cambiará dependiendo de la plataforma de ejecución (sistema operativo, entorno de ejecución [web, pc, celular, consola]) este patrón de diseño presenta una forma sencilla y versátil de implementarlos en una capa separada del resto de los módulos de la aplicación, de una manera sencilla, de forma que se facilite su uso y ampliación.

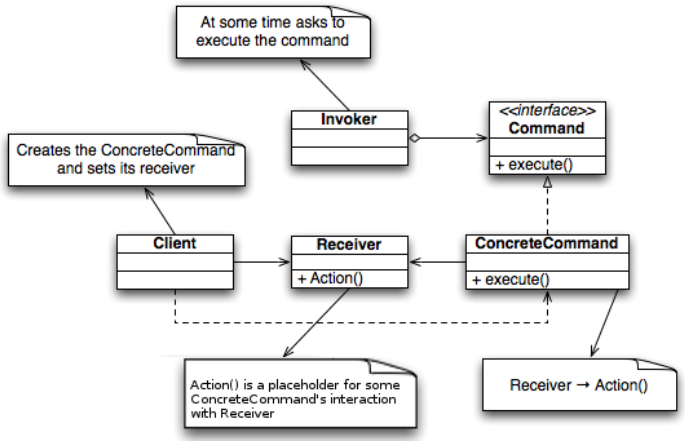
También, si necesita encolar comandos para ejecutarlos posteriormente en un orden, o para generar un sistema que permita deshacer y rehacer comandos, este patrón permite almacenar objetos de comando generando un historial de comandos y controlar el tiempo de su ejecución retrasando potencialmente una serie de acciones para su posterior reproducción. De manera similar, puede rehacerlos o deshacerlos y agregar flexibilidad adicional para controlar la ejecución de cada objeto de comando.

Aplicabilidad: Cualquier programa en el que se requiera facilitar la parametrización de las acciones a realizar o independizar el momento de petición del de ejecución y abstrayendo/desacoplando a estos elementos de la implementación de la orden. También es útil cuando se necesite implementar CallBacks, especificando que órdenes se requiere que se ejecuten en ciertas situaciones de otras órdenes. Es decir, un parámetro de una orden puede ser otra orden por ejecutar. Resulta especialmente útil cuando se necesita brindar el soporte "deshacer" o “rehacer”.

En el caso particular de los videojuegos:

- En un juego de estrategia en tiempo real, el patrón Command podría usarse para poner en cola acciones de unidades y edificios. Luego, el juego ejecutaría cada comando a medida que los recursos estuvieran disponibles.
- En un juego de estrategia por turnos, el jugador puede seleccionar una unidad y luego almacenar sus movimientos o acciones en una cola u otra colección. Al final del turno, el juego ejecutaría todos los comandos en la cola del jugador.
- En un juego de rompecabezas, el patrón Command podría permitir al jugador deshacer y rehacer acciones.
- En un juego de lucha, leer las pulsaciones de botones o los movimientos del gamepad en una lista de comandos específica podría resultar en combos y movimientos especiales.
- Facilitar la configuración de comandos, cambiando el “botón” que la ejecuta.

Estructura:



Participantes:

- Command: Define la interfaz de ejecución de comandos
- ConcreteCommand: Implementa la interfaz de ejecución de operaciones del receptor. De esta manera relaciona una acción con un receptor (Receiver)
- Client: Crea un comenado concreto e india a quien va dirigido (Receiver)
- Invoker: Contiene el comando asociado a la petición
- Receiver: Sabe como realizar las operaciones asociadas a una petición. Cualquier clase puede actuar como un receptor.

Consecuencias:

- Desacopla el objeto que invoca la operación del que sabe como llevarla a cabo.
- Los comandos son entidades de primer orden: se pueden manipular y extender como cualquier objeto.
- Se pueden crear macro-comandos (patrón Composite)
- Es fácil añadir nuevos comandos ya que no es necesario cambiar las clases existentes.

Implementación: Existen dos maneras de implementar los comandos:

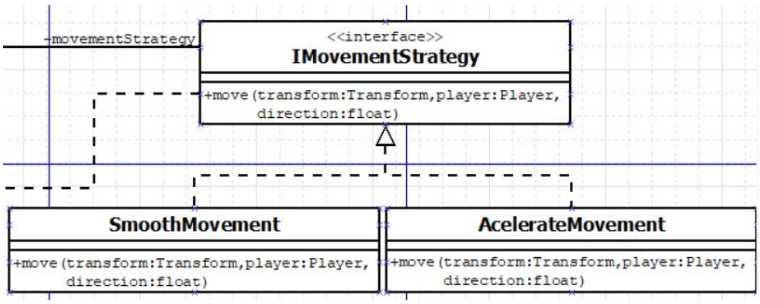
- Invocar una acción en el receptor
- Implementar el comportamiento sin delegar: útil para comandos independientes de un receptor, o cuando el receptor está implícito.

Existen diferentes maneras de implementar las operaciones deshacer y rehacer:

- Operaciones adicionales en Command
- Almacenar el estado previo a la ejecución del comando: Esto en el objeto receptor Considere que: almacenar únicamente el estado anterior permite realizar un único deshacer. Si almacena varios estados, puede realizar varios deshacer.

Considerando las necesidades de nuestro videojuego, el receptor está implícito en un Script que sea el encargado de determinar el Input usado. Por otro lado, no necesitamos implementar las operaciones deshacer y rehacer.

Entonces, nuestra propuesta es modificar el patrón Strategy para que reciba la dirección del movimiento



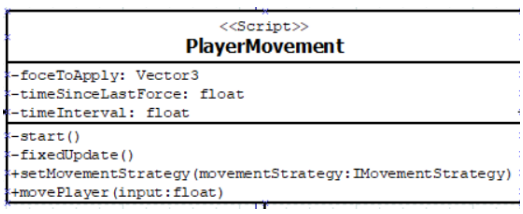
Las implementaciones quedarán así

```
using System;
using UnityEngine;

4 referencias
public interface IMovementStrategy
{
    2 referencias
    public void Move(Transform transform, Player player, float direction);
}
```

```
using System;
using UnityEngine;

1 referencia
public class SmoothMovement : IMovementStrategy
{
    2 referencias
    public void Move(Transform transform, Player player, float direction)
    {
        float moveInX = direction * player.Velocity * Time.deltaTime;
        transform.Translate(moveInX, 0, 0);
    }
}
```

```
using System;
using System.IO;
using UnityEngine;
using UnityEngine.Windows;

1 referencia
public class AcelerateMovement : IMovementStrategy
{
    2 referencias
    public void Move(Transform transform, Player player,float direction)
    {
        float movement = direction * (player.Velocity + player.Acceleration) * Time.deltaTime

        transform.Translate(movement * Time.deltaTime, 0, 0);
    }
}
```

De esta manera ya no tenemos el Input dentro de estas implementaciones. Esto provoca que el Script PlayerMovement genere un error, ya que el método move() necesita de un parámetro adicional. Entonces corregiremos esta situación quedando:

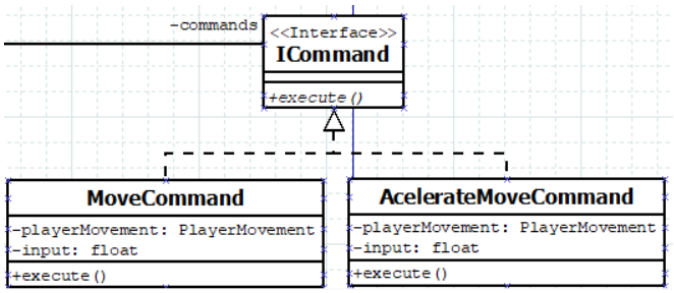


Es decir:

```
public void MovePlayer(float input)
{
    movementStrategy.Move(transform, player, input);
}
```

Note que además se ha eliminado el método Update() en el cual se invocaba constantemente al método MovePlayer(). Será necesario otro mecanismo para que se ejecute este método.

Ahora registremos cada input como un comando, para lo cual vamos a crear una carpeta command dentro de la carpeta model y en esta carpeta implementaremos los comandos siguiendo el siguiente esquema:



Empezando por la interface

```
using System;

public interface ICommand
{
    void Execute();
}
```

Y siguiendo por las implementaciones:

```
using System;

public class MoveCommand : ICommand
{
    private readonly PlayerMovement playerMovement;
    private readonly float input;

    public MoveCommand(PlayerMovement playerMovement, float input)
    {
        this.playerMovement = playerMovement;
        this.input = input;
    }

    public void Execute()
    {
        playerMovement.SetMovementStrategy(new SmoothMovement());
        playerMovement.MovePlayer(input);
    }
}
```

Observe como aquí se envía como parámetro del método MovePlayer, el valor del input

De manera similar

```
using System;
1 referencia
public class AcelerateMoveCommand : ICommand
{
    private readonly PlayerMovement playerMovement;
    private readonly float input;

    0 referencias
    public AcelerateMoveCommand(PlayerMovement playerMovement, float input)
    {
        this.playerMovement = playerMovement;
        this.input = input;
    }

    1 referencia
    public void Execute()
    {
        playerMovement.SetMovementStrategy(new AcelerateMovement());
        playerMovement.MovePlayer(input);
    }
}
```

Bien, finalmente, usaremos otro Script denominado PlayerController para la detección del evento de entrada (teclas) que se encargará de ejecutar el comando adecuado

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

Script de Unity | 0 referencias
public class PlayerController : MonoBehaviour
{
    private PlayerMovement playerMovement;
    private List<ICommand> commands;

    Mensaje de Unity | 0 referencias
    void Start()
    {
        playerMovement = gameObject.GetComponent<PlayerMovement>();
        commands = new List<ICommand>();
    }

    Mensaje de Unity | 0 referencias
    void Update()
    {
        commands.Clear();
        float horizontalInput = Input.GetAxis("Horizontal");
        commands.Add(new MoveCommand(playerMovement,horizontalInput));

        if (Input.GetKey(KeyCode.Space))
        {
            commands.Add(new AcelerateMoveCommand(playerMovement,horizontalInput));
        }

        foreach (var command in commands)
        {
            command.Execute();
        }

    }
}
```

De esta manera queda definido el diagrama de clases que presenta la historia de usuario

Conclusión

El patrón Strategy se usa para encapsular las diferentes estrategias de movimiento (SmoothMovementStrategy y AcceleratedMovementStrategy), permitiendo cambiar la forma en que el Player se mueve según las entradas del usuario.

El patrón Command Hace explícito el comando a ejecutar, permitiendo que la lógica de entrada (InputPlayerHandler) no esté directamente vinculada a la lógica de ejecución. Los comandos concretos (MoveCommand y AcceleratedMoveCommand) encapsulan la lógica específica de cada tipo de movimiento.

Esto provoca tanto ventajas como desventajas:

Método	Ventajas	Desventajas
Strategy	Modularidad, fácil extensión para nuevos tipos de movimiento.	Puede agregar complejidad si se utilizan demasiadas estrategias.
Command (explícito aquí)	Desacopla la entrada del usuario de la ejecución de comandos.	Requiere una mayor cantidad de clases y puede complicar la arquitectura.

Este enfoque hace que el código sea más flexible y escalable. Si en el futuro se necesitan nuevas formas de movimiento o comandos adicionales, se pueden agregar sin modificar el resto del sistema, facilitando el mantenimiento y la expansión del código.