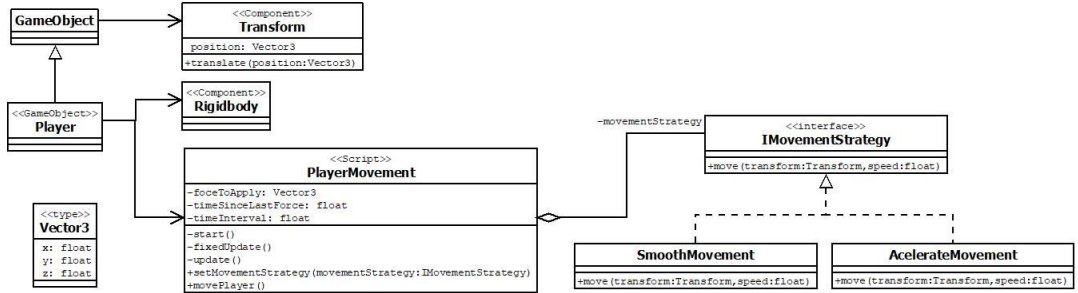


Historia de Usuario: Movimiento lateral del player	
Identificador: UH03	Usuario: Jugador
Prioridad en negocio: -	Riesgo en desarrollo: -
Puntos estimados: -	Iteración asignada: -
Programador Responsable: Ariel Vega	
<p>Descripción</p> <p>Como jugador quiero que el jugador pueda moverse hacia los laterales usando movimientos suaves o bruscos para poder aplicar diferentes estrategias según lo necesite.</p>	
<p>Observaciones</p> <p>Existe la posibilidad de que se puedan agregar otros tipos de movimientos laterales, o incluso nuevos movimientos. El diseño debe contemplar esto</p>	
<p>Diagrama de clases</p>  <pre> classDiagram class GameObject class Transform { <<Component>> position: Vector3 +translate(position: Vector3) } class Player { <<GameObject>> } class Rigidbody { <<Component>> } class Vector3 { <<Type>> x: float y: float z: float } class PlayerMovement { <<Script>> -forceToApply: Vector3 -timeSinceLastForce: float -timeInterval: float -start() -fixedUpdate() -update() +useMovementStrategy(movementStrategy: IMovementStrategy) +movePlayer() } class IMovementStrategy { <<Interface>> +move(transform: Transform, speed: float) } class SmoothMovement { +move(transform: Transform, speed: float) } class AccelerateMovement { +move(transform: Transform, speed: float) } GameObject < -- Player Transform < -- Rigidbody PlayerMovement o-- Rigidbody PlayerMovement o-- IMovementStrategy IMovementStrategy < .. SmoothMovement IMovementStrategy < .. AccelerateMovement </pre>	
<p>Criterios de aceptación</p> <ul style="list-style-type: none"> El movimiento lateral varía según el tipo de estrategia elegido. Se debe garantizar la posibilidad de agregar nuevas estrategias de movimiento de tal manera que haya bajo acoplamiento de clases. 	

Actividad 01: Aplicar el patrón de diseño Strategy para definir los tipos de movimientos en diferentes clases.

Los patrones de diseño son técnicas para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Existen diferentes categorías de patrones de diseño según su nivel de abstracción o el tipo de problema que soluciona. Respecto de esta última categoría podemos clasificarlos en:

- Patrones creacionales: Corresponden a patrones de diseño de software que solucionan problemas en la creación de instancias. Nos ayudan a encapsular y abstraer dicha creación.
- Patrones estructurales: Son los patrones de diseño software que solucionan problemas que surgen de la composición (agregación) de clases y objetos.
- Patrones de comportamiento: Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción y responsabilidades entre clases y objetos, así como los algoritmos que encapsulan.

El patrón de diseño Strategy, es un patrón de comportamiento porque determina cómo se debe realizar el intercambio de mensajes entre diferentes objetos para resolver una tarea. El patrón Strategy permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Los patrones de diseño generalmente son descriptos por medio de los siguientes elementos, que serán descriptos para el caso particular del Strategy:

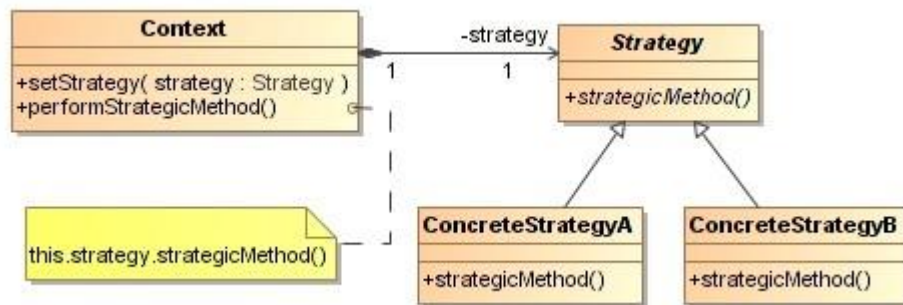
Propósito: Definir una familia de algoritmos, colocar cada uno de ellos en una clase separada (estrategia concreta) y hacer sus objetos intercambiables de manera dinámica según la necesidad del cliente.

Motivación: Si debe implementar la realización de una tarea de diferentes maneras; o si tiene la certeza de que esa tarea se podrá realizar en el futuro de diferente manera, resulta conveniente separar las clases encargadas de realizarlas (clientes) de los diferentes algoritmos que la realizan (estrategias), por diversos motivos:

- Incluir el código de los algoritmos en los clientes hace que estos sean demasiado grandes y complicados de mantener y/o extender.
- El cliente no va a necesitar todos los algoritmos en todos los casos, de modo que no queremos que dicho cliente los almacene si no los va a usar.
- Si existiesen clientes distintos que usasen los mismos algoritmos, habría que duplicar el código, por tanto, esta situación no favorece la reutilización.
- La solución que el patrón estrategia supone para este escenario pasa por encapsular los distintos algoritmos en una jerarquía y que el cliente trabaje contra un objeto intermediario contexto. El cliente puede elegir el algoritmo que prefiera de entre los disponibles, o el mismo contexto puede ser el que elija el más apropiado para cada situación.

Aplicabilidad: Cualquier programa que ofrezca un servicio o función determinada, que pueda ser realizada de varias maneras, es candidato para utilizar el patrón Strategy. Puede haber cualquier número de estrategias y cualquiera de ellas podrá ser intercambiada por otra en cualquier momento, incluso en tiempo de ejecución. Si existen clases relacionadas que se diferencian únicamente por su comportamiento probablemente admite el uso de Strategy. Si un algoritmo utiliza información que no deberían conocer los clientes, con Strategy evita la exposición de dichas estructuras. Si requiere definir múltiples comportamientos y desea evitar usar estructuras condicionales para elegir cual se aplicará puede usar Strategy para encapsular estos comportamientos en clases concretas.

Estructura:



Participantes:

- **Context:** Es el elemento o cliente que usa los algoritmos; y que delega en la jerarquía de estrategias el encapsulamiento de estos algoritmos. Establece una estrategia concreta mediante una referencia a la estrategia necesaria por medio del método `setStrategy()`, el cual recibe como parámetro la estrategia concreta. El método `performStrategicMethod()` se encarga de ejecutar la estrategia.
- **Strategy:** Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el **Context** para invocar a la estrategia concreta. A nivel de implementación puede ser una interface o una clase abstracta.
- **ConcreteStrategy:** Implementa el algoritmo utilizando la interfaz definida por la estrategia a través del método `strategicMethod()`.

Consecuencias:

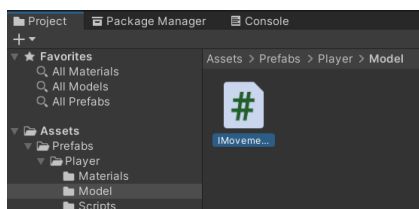
- La herencia puede ayudar a factorizar las partes comunes de las familias de algoritmos (sustituyendo el uso de bloques de instrucciones condicionales). En este contexto es común la aparición conjunta de otros patrones como el patrón **Template**.
- El uso del patrón proporciona una alternativa a la extensión de contextos, ya que puede realizarse un cambio dinámico de estrategia.
- Los clientes deben conocer las diferentes estrategias y debe comprender las posibilidades que ofrecen.
- Como contrapartida, aumenta el número de objetos creados, por lo que se produce una penalización en la comunicación entre estrategia y contexto (hay una indirección adicional).

Implementación: Entre las posibilidades disponibles a la hora de definir la interfaz entre **Strategy** y **Context**, puede considerarse:

- Pasar como parámetro la información necesaria para la estrategia implica un bajo acoplamiento y la posibilidad de envío de datos innecesarios.
- Pasar como parámetro el contexto y dejar que la estrategia solicite la información que necesita supone un alto acoplamiento entre ellos.
- Mantener en la estrategia una referencia al contexto (similar al anterior).

- También puede ocurrir que se creen y se utilicen los objetos Strategy en el contexto solo si es necesario, en tal caso las estrategias serán opcionales.

Con lo anterior se define una interface denominada IMovementStrategy (previamente en la carpeta Player crear una subcarpeta denominada Model; en esta carpeta generar la interface). Quedará similar a lo siguiente:



Y el contenido de este archivo es

```
1 using System;
2 using UnityEngine;
3
4 public interface IMovementStrategy
5 {
6     public void Move(Transform transform, float speed);
7 }
```

Actividad 02: Crear la primera estrategia concreta. Se debe crear una clase que implemente la interface IMovementStrategy

```
1 using System;
2 using UnityEngine;
3
4 public class SmoothMovement : IMovementStrategy
5 {
6     public void Move(Transform transform, float speed)
7     {
8         throw new NotImplementedException();
9     }
10 }
```

Ahora se implementa el método

```
1 using System;
2 using UnityEngine;
3
4 public class SmoothMovement : IMovementStrategy
5 {
6     public void Move(Transform transform, float speed)
7     {
8         float moveInX = Input.GetAxis("horizontal") * speed * Time.deltaTime;
9         transform.Translate(moveInX, 0, 0);
10     }
11 }
```

Observe que se obtiene el movimiento horizontal a través de la clase Input; y se utiliza este valor para determinar el movimiento en conjunto con la velocidad recibida por parámetro y la aplicación del deltaTime.

Actividad 03: Probar la estrategia concreta. Para lograr esto, debemos plantear los cambios necesarios en el Script PlayerMovement

En primer lugar definimos un atributo que represente la estrategia y la inicializamos en el método Start()

```

1  using UnityEngine;
2
3  Script de Unity (1 referencia de recurso) | 0 referencias
4  public class PlayerMovement : MonoBehaviour
5  {
6      private Vector3 forceToApply;
7      private float timeSinceLastForce;
8      private float intervalTime;
9
10     private Vector3 speed;
11     private IMovementStrategy movementStrategy;
12     Mensaje de Unity | 0 referencias
13     private void Start()
14     {
15         forceToApply = new Vector3(0, 0, 10);
16         timeSinceLastForce = 0f;
17         intervalTime = 2f;
18
19         speed = new Vector3(5f, 0, 0);
20         SetMovementStrategy(new SmoothMovement());
21     }

```

Puede notar que el tipo del atributo movementStrategy es la interface IMovementStrategy. Además en la línea 18 para establecer la estrategia concreta se invoca al método SetMovementStrategy(). En este método se envía como parámetro la estrategia concreta.

```

21  1 referencia
22  public void SetMovementStrategy(IMovementStrategy movementStrategy)
23  {
24      this.movementStrategy = movementStrategy;
25  }

```

Como puede notar es una simple setter. Por otro lado, en Update() se invoca el método MovePlayer() para que constantemente se esté verificando si el jugador se está moviendo.

```

26  Mensaje de Unity | 0 referencias
27  private void Update()
28  {
29      MovePlayer();
30  }
31
32  1 referencia
33  public void MovePlayer()
34  {
35      movementStrategy.Move(transform, speed.x);
36  }

```

El método MovePlayer() invoca el método Move() de la estrategia. De esta manera por el polimorfismo, se ejecutará la estrategia concreta aplicada. Observe, además, que se envía como argumentos el transform del GameObject, y la velocidad definida como atributo en el Script.

Si prueba el proyecto debería notar que puede mover el gameObject hacia los costados con una velocidad constante.

Actividad 04: Programar el movimiento brusco o acelerado. En este caso se necesita únicamente crear una clase que implemente la interface. Por ejemplo, en la siguiente implementación se puede observar que se mantiene un movimiento lateral acelerado entre un valor máximo y un valor mínimo

```

1  using System;
2  using UnityEngine;
3
4  0 referencias
5  public class AccelerateMovement : IMovementStrategy
6  {
7      private float currentSpeed = 0f;
8      private float acceleration = 2f;
9
10     2 referencias
11     public void Move(Transform transform, float speed)
12     {
13         currentSpeed += Input.GetAxis("Horizontal") * acceleration * Time.deltaTime;
14         currentSpeed = Mathf.Clamp(currentSpeed, -speed, speed);
15         transform.Translate(currentSpeed * Time.deltaTime, 0, 0);
16     }
17 }

```

Para probar esta estrategia en el Script se lo instancia el lugar de la estrategia anterior:

```

private void Start()
{
    forceToApply = new Vector3(0, 0, 10);
    timeSinceLastForce = 0f;
    intervalTime = 2f;

    speed = new Vector3(5f, 0, 0);
    //SetMovementStrategy(new SmoothMovement());
    SetMovementStrategy(new AccelerateMovement());
}

```

Actividad 05: Separar la capa del patrón. Si observa el prototipo desarrollado, AccelerateMovementStrategy define la velocidad actual y la aceleración, mientras que SmoothMovementStrategy recibe la velocidad. Estos elementos son propios de una clase, por ejemplo, Player. Incluyamos esta clase, y luego aplicaremos los cambios necesarios

```

1  using System;
2
3  1 referencia
4  public class Player
5  {
6      private float velocity;
7      private float acceleration;
8
9      0 referencias
10     public Player(float velocity, float acceleration)
11     {
12         this.velocity = velocity;
13         this.acceleration = acceleration;
14     }
15
16     0 referencias
17     public float Velocity { get => velocity; set => velocity = value; }
18     0 referencias
19     public float Acceleration { get => acceleration; set => acceleration = value; }
20 }

```

Vamos a definir dentro del Script PlayerMovement un atributo que represente a un objeto de esta clase

```

1  using UnityEngine;
2
3  Script de Unity (1 referencia de recurso) | 0 referencias
4  public class PlayerMovement : MonoBehaviour
5  {
6      private Vector3 forceToApply;
7      private float timeSinceLastForce;
8      private float intervalTime;
9      private IMovementStrategy movementStrategy;
10     private Player player;
11
12     Mensaje de Unity | 0 referencias
13     private void Start()
14     {
15         forceToApply = new Vector3(0, 0, 10);
16         timeSinceLastForce = 0f;
17         intervalTime = 2f;
18         player = new Player(5f, 5f);
19         //SetMovementStrategy(new SmoothMovement());
20         SetMovementStrategy(new AccelerateMovement());
21     }
22 }

```

Se observa que en la línea 9 se genera un atributo de tipo Player. En la línea 16 se instancia este atributo. Observe además que se elimina la variable speed del Script ya que se usará el valor del atributo velocity de player.

Esto genera un error en el método Move() ya que no existe un parámetro speed de tipo float. Vamos a modificar este método para que reciba un Player

```
1  using System;
2  using UnityEngine;
3
4  4 referencias
5  public interface IMovementStrategy
6  {
7      1 referencia
8      public void Move(Transform transform, Player player);
9  }
```

Entonces en SmoothMovementStrategy queda

```
1  using System;
2  using UnityEngine;
3
4  0 referencias
5  public class SmoothMovement : IMovementStrategy
6  {
7      2 referencias
8      public void Move(Transform transform, Player player)
9      {
10         float moveInX = Input.GetAxis("Horizontal") * player.Velocity * Time.deltaTime;
11         transform.Translate(moveInX, 0, 0);
12     }
13 }
```

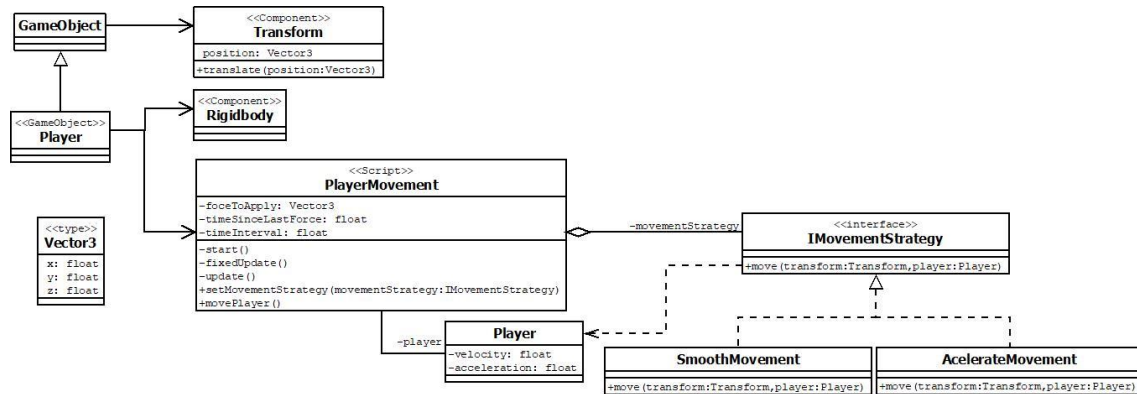
Y AccelerateMovementStrategy queda

```
1  using System;
2  using UnityEngine;
3
4  1 referencia
5  public class AccelerateMovement : IMovementStrategy
6  {
7      private float currentSpeed = 0f;
8
9      2 referencias
10     public void Move(Transform transform, Player player)
11     {
12         currentSpeed += Input.GetAxis("Horizontal") * player.Acceleration * Time.deltaTime;
13         player.Velocity = Mathf.Clamp(currentSpeed, -player.Velocity, player.Velocity);
14         transform.Translate(currentSpeed * Time.deltaTime, 0, 0);
15     }
16 }
```

Con esto podemos corregir el Script PlayerMovement

```
1  1 referencia
2  public void MovePlayer()
3  {
4      movementStrategy.Move(transform, player);
5  }
```

Considere que debe actualizar el Diagrama de clase en la Historia de Usuario, quedando de esta manera



Conclusión

El patrón Strategy ofrece varios beneficios en el desarrollo de videojuegos:

- **Flexibilidad:** Permite cambiar dinámicamente el comportamiento del movimiento lateral sin modificar la estructura principal del código.
- **Mantenimiento:** Al encapsular los algoritmos de movimiento en clases separadas, el código es más fácil de mantener y extender.
- **Reusabilidad:** Las estrategias pueden ser reutilizadas en otros proyectos o en diferentes contextos dentro del mismo juego.

El uso del patrón de diseño Strategy para el movimiento lateral de un GameObject en Unity no solo mejora la modularidad y reusabilidad del código, sino que también facilita la implementación de diferentes estilos de movimiento sin comprometer la estructura general del proyecto.