Proyecto DDS

Videojuego

Descripción del proyecto

El proyecto consiste en el desarrollo de un videojuego de estrategia por turnos **TBS** (**T**urn **B**ased **S**trategy), nos hemos inspirado en el famoso juego de *Advance Wars* de Nintendo para para Game Boy Advance, podríamos decir que el juego es como el ajedrez, pero con unas pocas diferencias:

- En un mismo turno un jugador puede mover diversas unidades (piezas).
- Una unidad (pieza) puede moverse dentro de un rango, es decir no tiene que moverse siempre la misma distancia, si no que puede variar dependiendo del deseo del jugador.
- Una unidad no puede pasar por encima de otra unidad, ya sea aliada o enemiga.
- Si después de mover una unidad a un nuevo sitio tiene en rango de ataque una unidad enemiga, la unidad puede atacar a esta o esperar.
- La forma de "luchar" entre piezas es más compleja que el ajedrez, en este caso las unidades tienen un daño y una vida concreta, y es necesario que su vida se vea reducida a 0 para que la unidad muera.
- El jugador posee fábricas donde puede construir nuevas unidades
- Para ganar el juego hay que acabar con todas las unidades enemigas O capturar la base enemiga (ESTO ÚLTIMO NO IMPLEMENTADO)

2- Tecnología utilizada: Unity

Para la realización de nuestro proyecto hemos decidido empelar ***Unity,*** un motor gráfico multiplataforma para videojuegos creado por Unity Technologies.

Los motores gráficos empleados por *Unity son*:

Sistema Operativo	Motor Gráfico
Windows	Direct3D
Linux, Mac	OpenGL
Android, iOS	OPENGL ES
Otros	Interfaces Propietarias

Unity tiene una interfaz amigable y fácil de usar, simplifica el trabajo con archivos, modificar los objetos del mundo es algo muy simple, también incluye herramientas para modificar código, audio, imágenes, control de versiones, etc, así como proporcionar ayuda para los programadores incluyendo funciones tales como calculo de costes de caminos, movimientos de objetos...

Cabe mencionar que Unity admite una gran cantidad de formatos de modelo así como sus respectivos programas, incluso llegando a interaccionar con estos para la edición del modelo desde Unity.

En cuanto a la programación, Unity permite la programación tanto en C# de .Net como javaScript, Boo o incluso UnityScript, un lenguaje desarrollado explícitamente para su plataforma.

3- Problemas con la tecnología

Primero vamos a definir el contexto, hemos trabajado sobre C# y hay que mencionar que este lenguaje **no permite herencia múltiple.**

- **Problemas de Herencia:** Para que los objetos del motor de Unity tengan una clase asociada, esta clase debe heredar de la clase **MonoBehaviour**. Esto nos ha causado diversos problemas a la hora de aplicar patrones, ya que hemos tenido que emplear interfaces en vez de clases abstractas en algunos casos.
- **No está permitido hacer *new* de clases heredadas de *MonoBehaviour*:** Esto ha supuesto que en el patrón de fábrica hayamos tenido que hacer una fábrica de prototipos en vez de una fábrica normal
- **Algunas funciones solo funcionan si la clase hereda de *MonoBehaviur*:** Esto ha supuesto problemas porque por ejemplo, no se puede crear una interfaz si no se hereda de esta clase (problemas con patrón estado), mover un objeto (problemas con patrón estado), o copiar un prototipo (problema con patrón fábrica)
- **Problemas con el entorno de desarrollo de código Unity**: Si se utiliza el sistema de Unit Test del entorno de desarrollo, el programa en general da error de compilación.

4- Diseño Arquitectónico

El diagrama de clases completo puede verse en el archivo adjunto. No lo hemos insertado dentro del documento porque es demasiado grande y no se apreciarían todos los detalles.

A lo largo del proyecto tenemos unas clases que heredan de la clase MonoBehaviour y otras que no. Las que sí heredan nos permiten utilizar los métodos new, OnGUI, Update, Start.. que serán determinantes en la ejecución en tiempo real. Por otra parte, las clases que no heredan de MonoBehaviour no podrán manejar este tipo de contextos.

Las siguientes clases NO heredan de MonoBehaviour:

- **Mapa**: Clase que se encarga de manejar el terreno y las unidades. Tiene una variable IColega, a través de la cual implementa un patrón Mediador con el que puede modificar los objetos Edificio, Terreno, Unidad, etc. Más adelante hablaremos en profundidad de este patrón. Para asegurarnos que sólo existe un objeto Mapa en la ejecución, le hemos aplicado también un patrón Singleton.
- **Turno**: Será la clase que inicialice el Mapa. Esta clase se encarga de identificar el turno en que nos encontramos y llevar la lista de jugadores, así como su orden.
- **Camino**: Clase que se encarga de calcular el camino más corto para mover una unidad a un punto determinado. Para ello utilizará un algoritmo de tipo A*.
- **EstadoUnidad**: Sirve como interfaz para los distintos estados por los que puede pasar el movimiento de una unidad. Utilizará el patrón Estado.

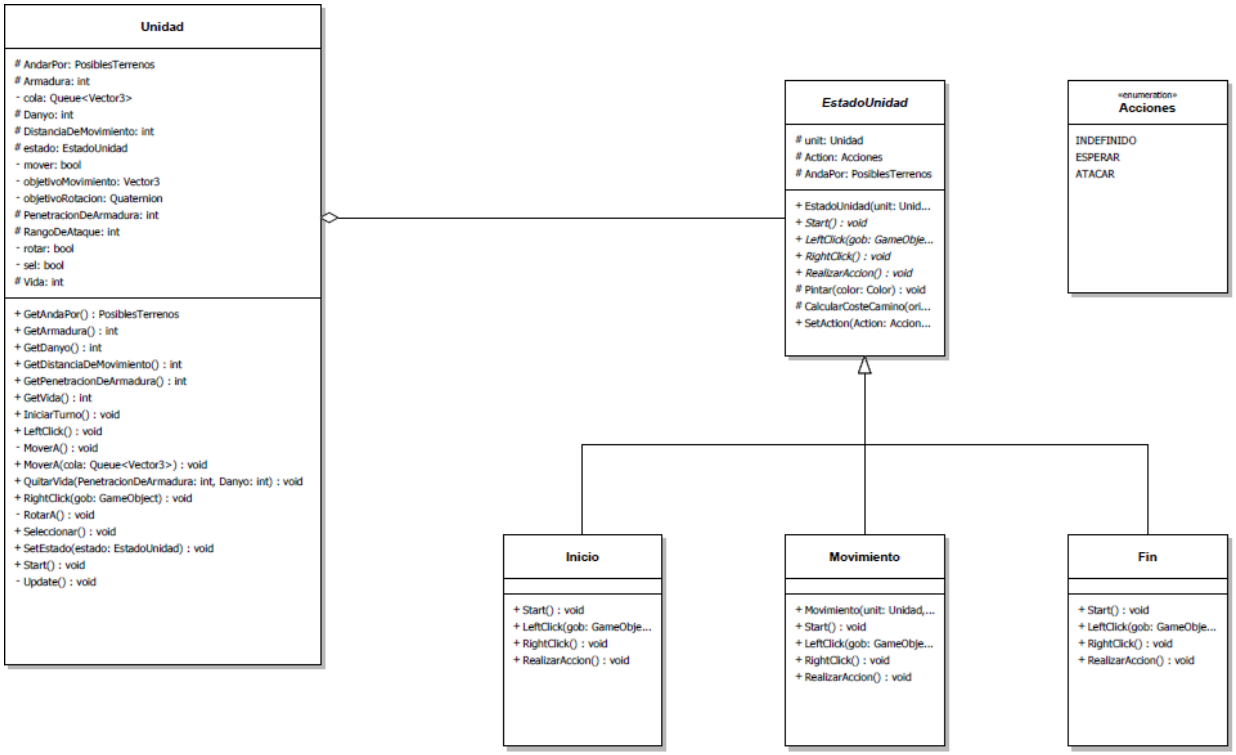
Las clases que SÍ heredan de MonoBehaviour son:

- **Player**: Una instancia para cada jugador. Contiene la información del dinero y el color de su equipo.
- **UserInput**: Esta clase es un ejemplo de la importancia de MonoBehaviour: es la encargada de recoger los pulsos de teclas y botones de ratón para traducirlos en eventos que realizarán determinadas acciones. Su método Update se ejecutará en bucle durante todo el tiempo que el objeto esté inicializado. Otros métodos importantes de la clase UserInput son los MoveCamera, que nos permitirán mover la cámara con botones o con el ratón. También están los métodos LeftClick y RightClick, que será reimplementados y heredados por toda la estructura de clases para definir qué sucede cuando pulsamos esos botones.
- **ControlInterfaz**: La clase que manejará las interfaces gráficas. Creará cuadros de texto, menús y botones.
- **WorldObjet**: La clase principal de la que heredarán los objetos Edificio y Unidad, los cuales irán reimplementando los métodos LeftClick, RightClick, etc.
- **Fabrica**: Clase de la que heredarán otras fábricas. Así mismo, utilizarán el patrón del mismo nombre para crear las distintas unidades.

5- Patrones de diseño aplicados

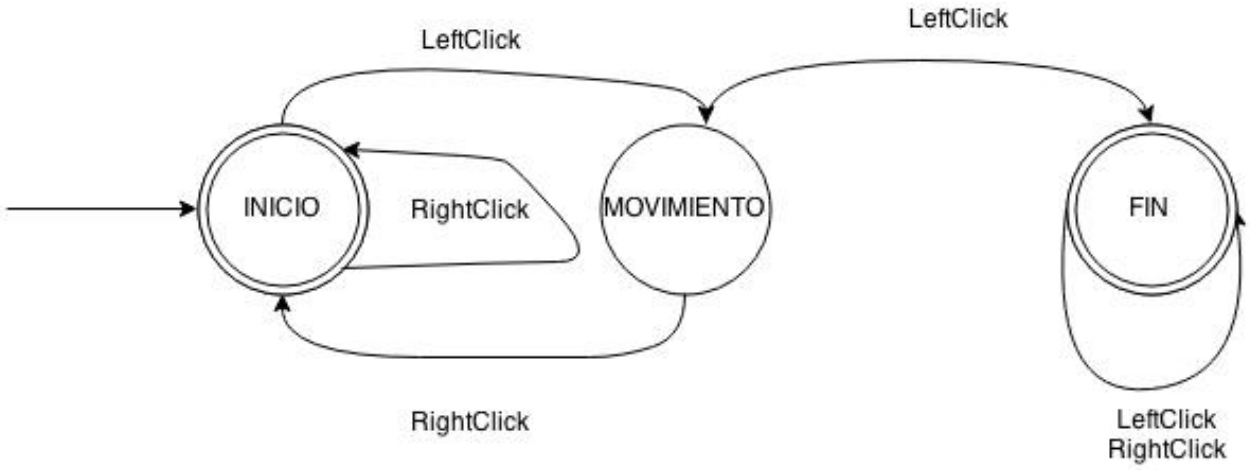
Estado

En nuestro juego manejamos unas unidades que por el tablero pudiendo realizar todo tipo de acciones. Para no sobrecargar la clase Unidad, hemos delegado los métodos referentes al movimiento y otras acciones en otras clases basándonos en el patrón Estado.



Ahora la clase Unidad tiene un objeto EstadoUnidad que podrá especificarse en tres tipos: Inicio, Movimiento y Fin. Con ellos hemos creado algo similar a una máquina de estados finitos. Las diferentes sobreescrituras de los métodos LeftClick() y RightClick() harán que avancemos y retrocedamos en estos estados.

- Inicio: La clase en la que se inicia el movimiento. Su reescritura del método LeftClick() nos dará el cálculo del camino que la unidad recorrerá. El método RightClick() cancelará el movimiento.
- Movimiento: La clase en la que se encuentra justo después de ejecutar el movimiento. LeftClick() comprueba si hay otras unidades alrededor, y si las hay realizará una acción, que vendrá determinada por el botón que hayamos presionado en la interfaz (esperar o atacar). El RightClick() hará que deshagamos el camino hecho en el estado anterior.
- Fin: Movimiento completado después de hacer alguna acción. Tanto LeftClick() como RightClick() terminan la acción.

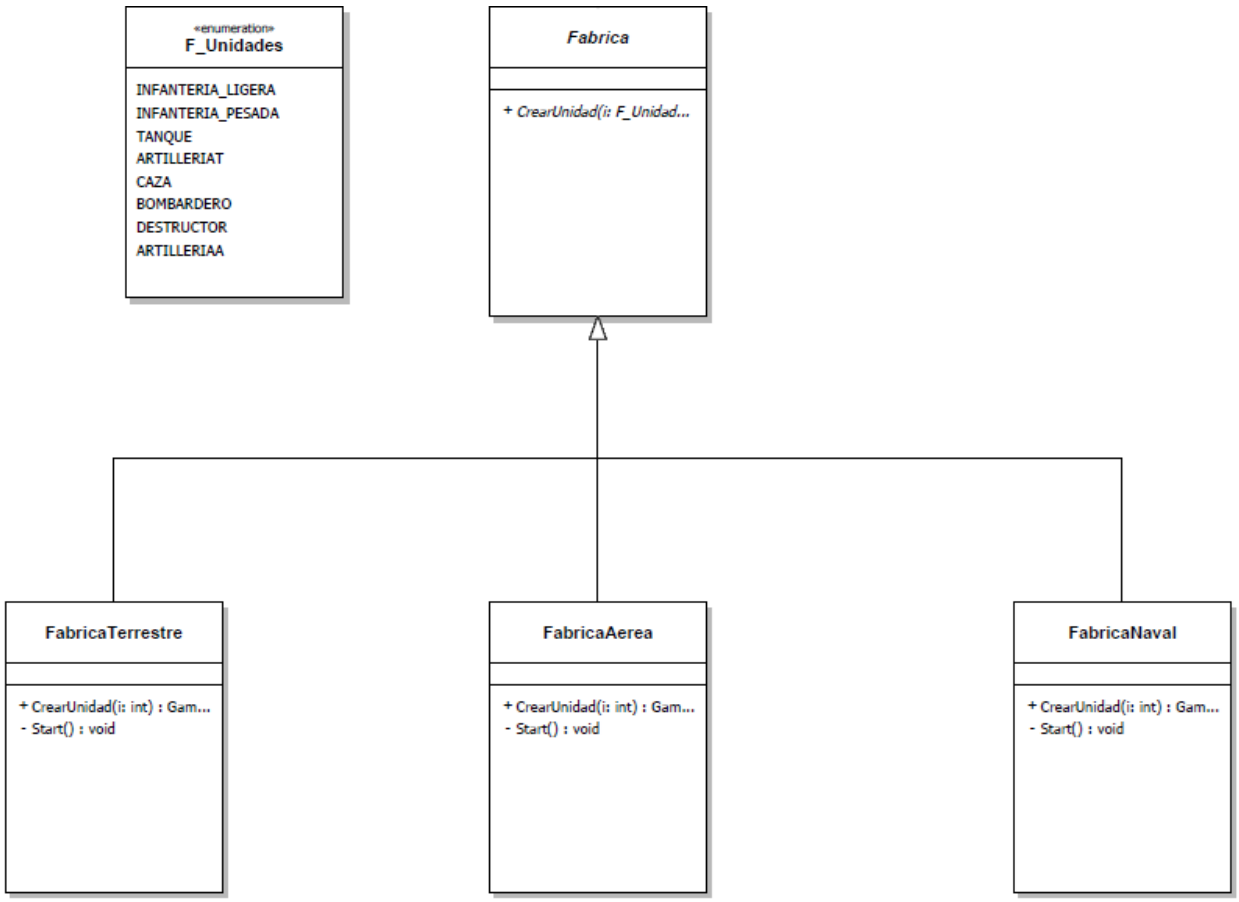


Ventajas: Mayor modularidad de las acciones que puede hacer una Unidad, y por tanto, mayor facilidad para navegar entre los estados, aparte de que nos evita estados inconsistentes. También nos ha permitido ahorra muchas sentencias condicionales, sobretodo de tipo switch.

Desventaja: Dado que la estructura del patrón estado no nos permite que las clases deriven de MonoBehaviour, nos priva de muchas funciones en tiempo real, haciendo que estas clases tengan qu e ser usadas por otras que si pueden usar métodos de MonoBehaviour.

Fábrica

Aquí hemos definido unos objetos llamados Fábricas que en el tablero del juego crearán otras unidades. Para hacerlas nos hemos basado en el patrón con el mismo nombre.



Tal y como sucede en el patrón Fábrica, de la clase Fabrica principal derivan todas las fábricas que específicas. Cada una de ellas implementarán unas unidades concretas, (Terrestre: infantería, tanques y artillería; Aérea: cazas y bombarderos; Naval: destructor y artillería). Esto es posible gracias a que las clases Unidad y sus derivadas tiene una estructura paralela a Fabrica.

Los distintos script Fabrica se asignan a los objetos Fabrica que hay en el mapa. Además, también tienen asignados una clase Edificio, que será la encargada de llamar a la Fabrica.

Ventajas: Simplificación del proceso de creación de objetos.

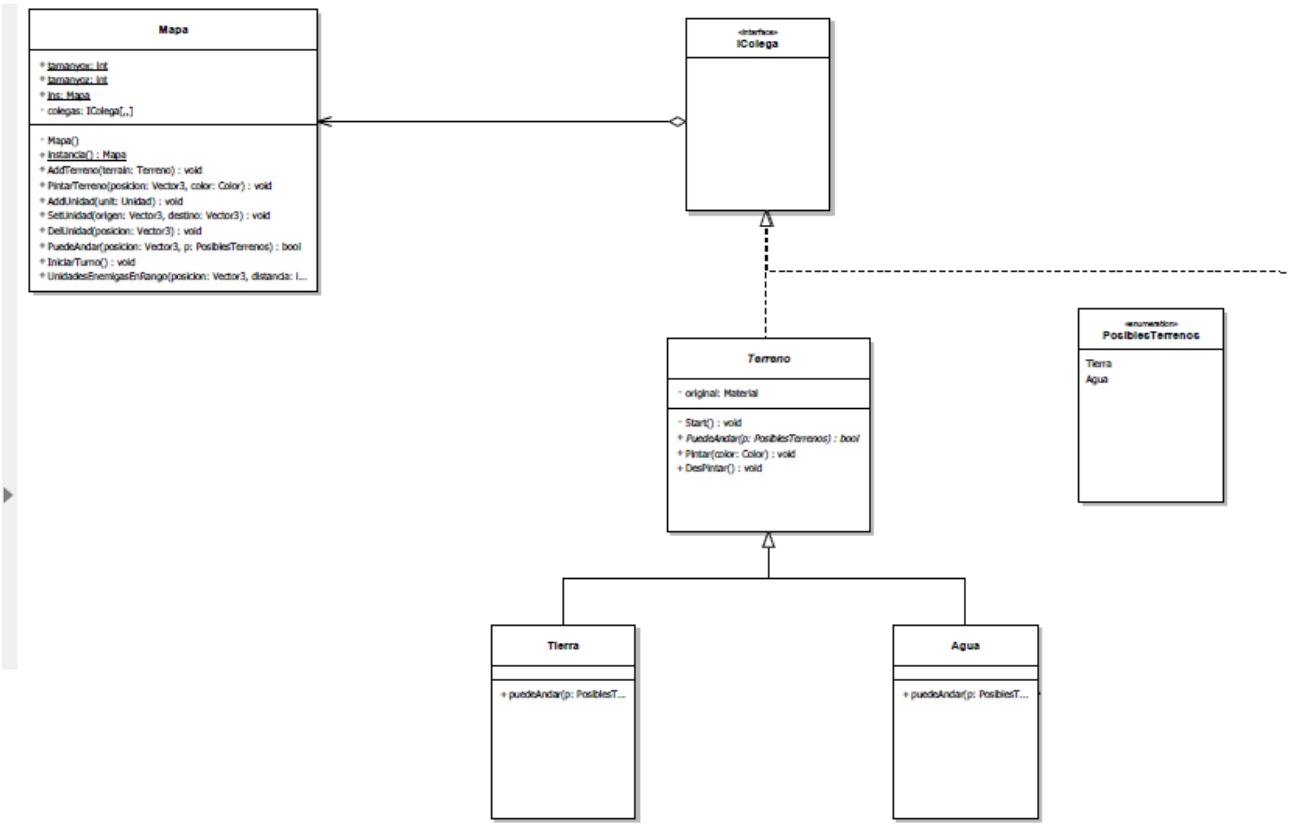
Desventajas:

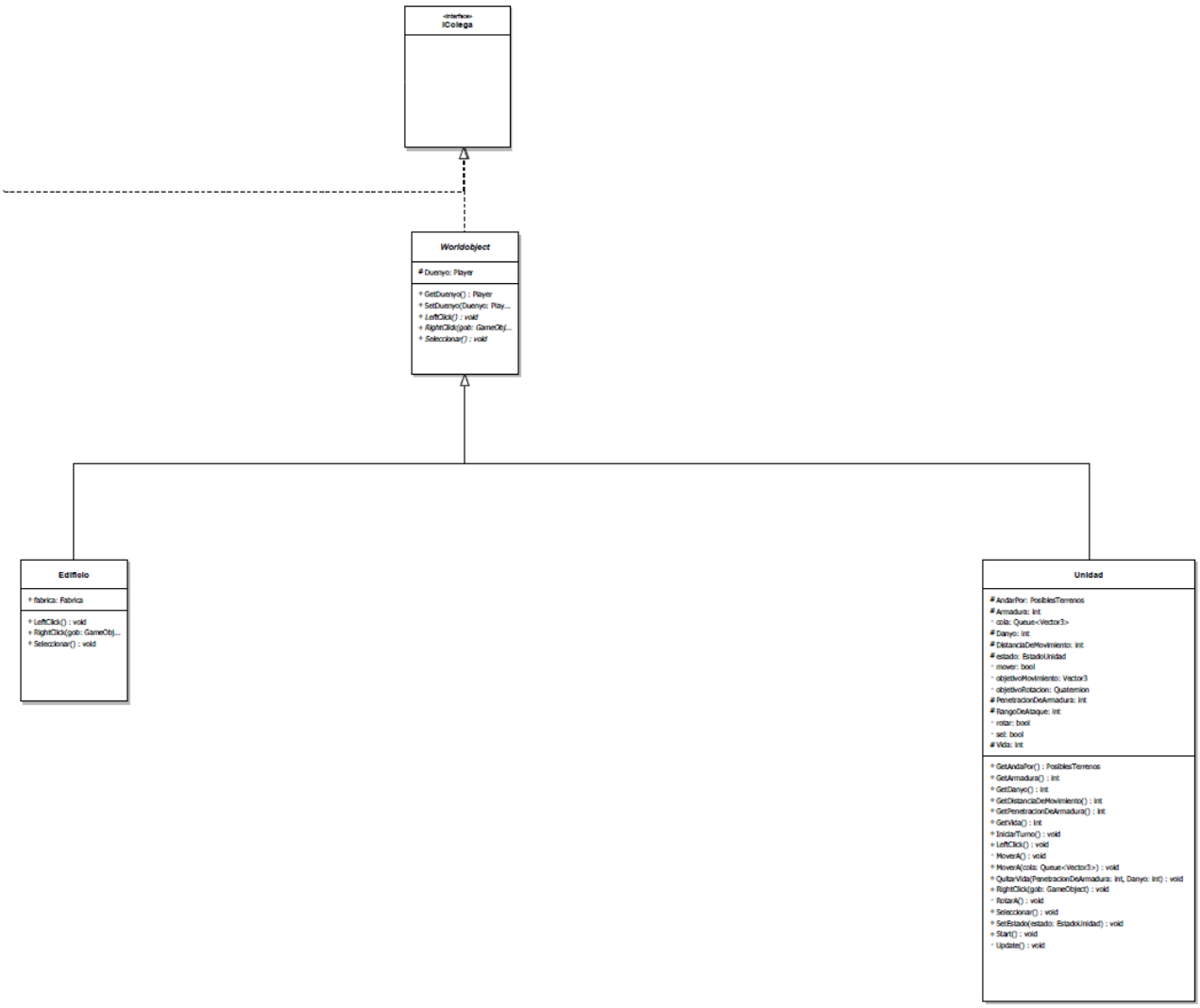
Mediador

En nuestro juego manejamos objetos WolrdObject, que podrán ser tanto Edificios como Unidades. Estos objetos interactúan con el Terreno, moviéndose, verificando su posición, creación del mapa, etc. Para no delegar sus responsabilidades en una única clase, hemos creado una interfaz en medio, tal y como sugiere el patrón medidador.

La clase principal será Mapa. Esta clase será la que tendrá un array IColega cuya instancia es una variable llamada 'colegas'. La razón de que esto será un array es que manejará el escenario, por lo que dos de sus componentes serán la longitud y la anchura. Cada uno de los cuadros del escenario se trata como un objeto, por lo que heredarán de esta interfaz mediadora.

Los herederos de WorldObject también implementan la interfaz IColega. Tanto Unidad como Edificio heredan de ella, y se crearán sus objetos a través de Mapa usando la interfaz mediadora 'colegas'.





Ventajas: Evita que la clase Unidad se comunique con la clase Terreno directamente (abstrae cooperación). Permite que los colegas cambien de posición (las unidades cambian su posición según se mueven).

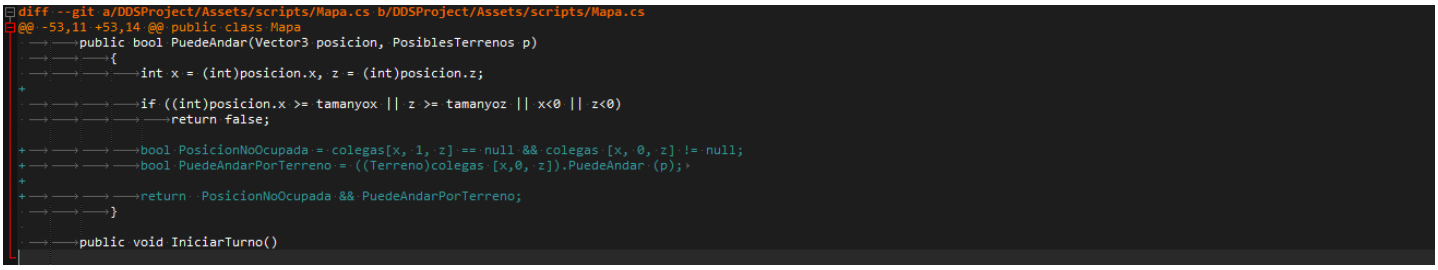
Desventajas: La tecnología no nos permite hacer new por lo que hemos tenido que hacer el mediador un singleton. Ya que C# no permite la multiherencia y todas las clases deben heredar de una misma tecnología por parte de la tecnología, hemos tenido que crear una interfaz Colega en vez de una clase abstracta.

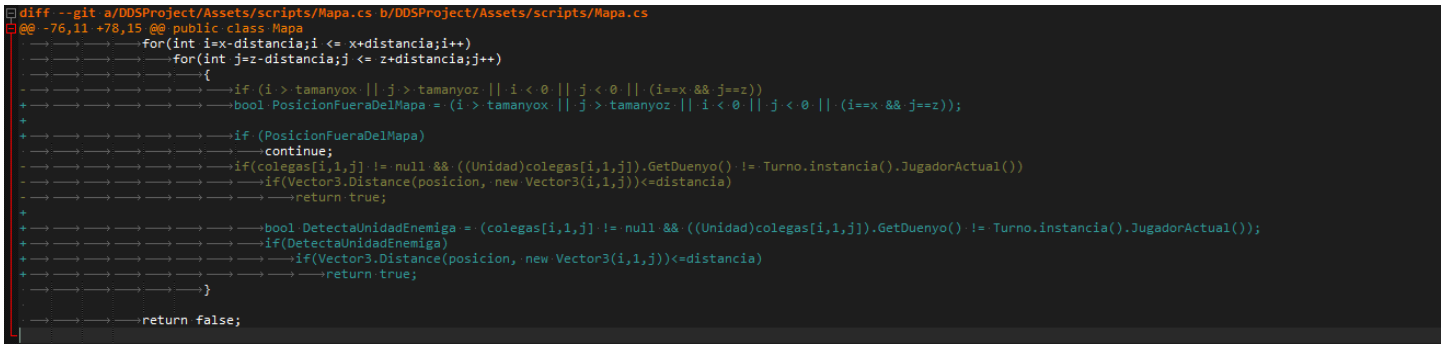
6- Refactorización aplicada

Una vez completada la funcionalidad del videojuego nos centramos en intentar hacer nuestro código más legible, mantenible… Es decir, en hacer *refactoring.* Las refactorizaciones aplicadas han sido las siguientes:

- Simplifying Conditional Expressions

Es posible que la o las condiciones de un if sean largas y anti-intuitivas, por lo que tengamos un código difícilmente legible y mantenible. Hemos seleccionado tres ocurrencias de esto dentro de nuestro código que considerábamos podían ser moradas para así clarificar nuestro código.



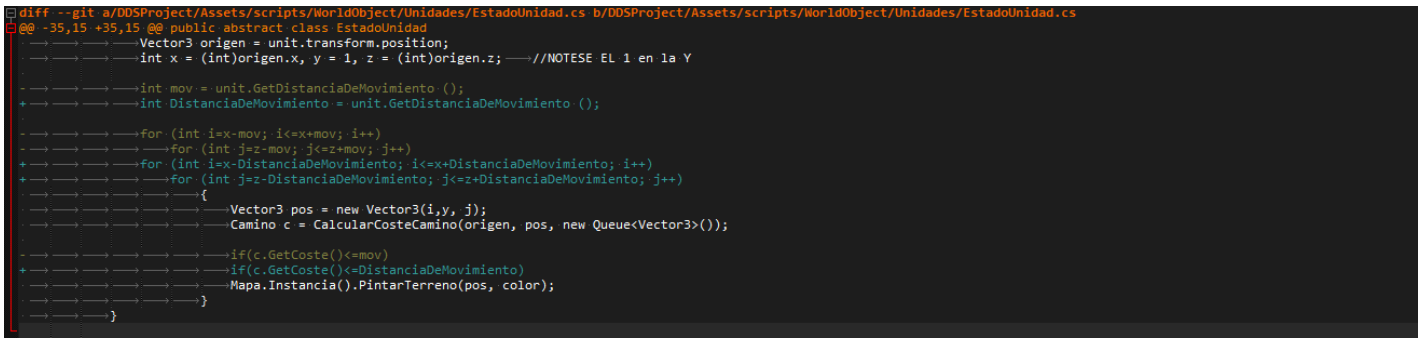


- # Rename Method

En aras de una mejor legibilidad y evitar posibles confusiones se modifica el nombre delk método “UnidadesEnRango” por “UnidadesEnemigasEnRango”. De este modo queda fuera de toda duda que el método hace referencia únicamente a las unidades enemigas y no a las aliadas y enemigas, como se podía deducir en un primer momento.

- # Rename Variable

Un ejemplo de tipo de refactoring consiste en renombrar las variables para que en un futor sea más fácil entender el código.

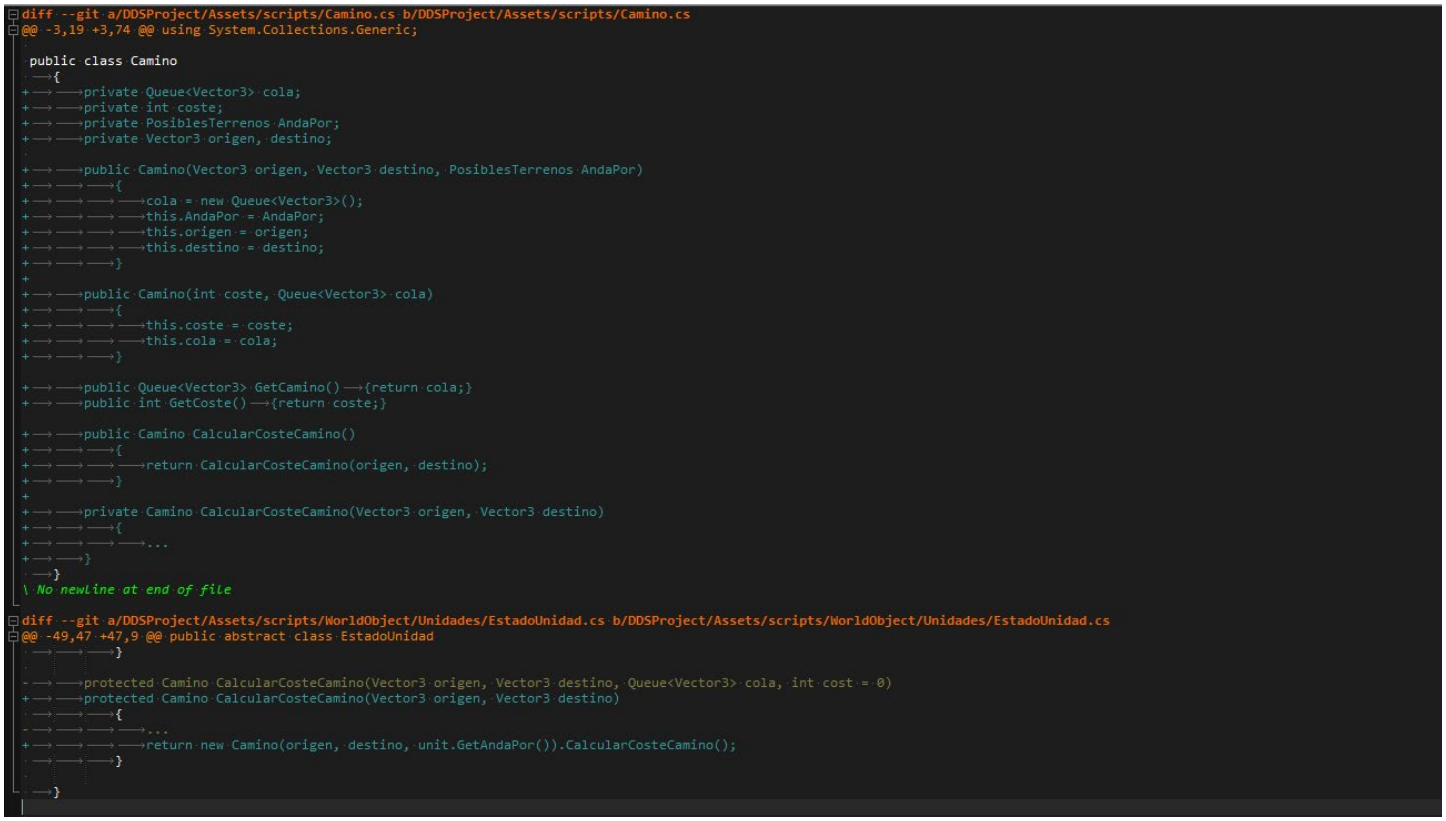


- # Consolidate Conditional Expression

Anteriormente teníamos dos expresiones condicionales independientes las cuales devolvían lo mismo. Se ha optado por unirlas en un solo if con una disyunción. Obtenemos así un código más limpio y claro.

- Replace Method With Method Object

El método para calcular el camino y su coste era largo y complejo, además de estar dividido en varias clases con sus variables correspondientes. Se ha optado pues, por crear una clase a partir del método, encapsulando así su comportamiento y variables y mejorando la modularidad y reutilización. Podemos observar una disminución de líneas en las demás clases.



```
diff --git a/DDSProject/Assets/scripts/Camino.cs b/DDSProject/Assets/scripts/Camino.cs
@@ -3,19 +3,74 @@ using System.Collections.Generic;

 public class Camino
 {
+    private Queue<Vector3> cola;
+    private int coste;
+    private PosiblesTerrenos AndaPor;
+    private Vector3 origen, destino;
+
+    public Camino(Vector3 origen, Vector3 destino, PosiblesTerrenos AndaPor)
+    {
+        cola = new Queue<Vector3>();
+        this.AndaPor = AndaPor;
+        this.origen = origen;
+        this.destino = destino;
+    }
+
+    public Camino(int coste, Queue<Vector3> cola)
+    {
+        this.coste = coste;
+        this.cola = cola;
+    }

+    public Queue<Vector3> GetCamino() {return cola;}
+    public int GetCoste() {return coste;}

+    public Camino CalcularCosteCamino()
+    {
+        return CalcularCosteCamino(origen, destino);
+    }
+
+    private Camino CalcularCosteCamino(Vector3 origen, Vector3 destino)
+    {
+        ...
+    }
 }
\ No newline at end of file

diff --git a/DDSProject/Assets/scripts/WorldObject/Unidades/EstadoUnidad.cs b/DDSProject/Assets/scripts/WorldObject/Unidades/EstadoUnidad.cs
@@ -49,47 +47,9 @@ public abstract class EstadoUnidad
         }

-        protected Camino CalcularCosteCamino(Vector3 origen, Vector3 destino, Queue<Vector3> cola, int cost = 0)
+        protected Camino CalcularCosteCamino(Vector3 origen, Vector3 destino)
         {
-            ...
+            return new Camino(origen, destino, unit.GetAndaPor()).CalcularCosteCamino();
         }

 }
```

7- Pruebas Unitarias

Hemos de mencionar que hemos tenido algunos problemas con las pruebas unitarias por culpa del entorno de desarrollo ya que cada vez que lo cerramos y lo volvemos a abrir se borran las librerías de referencia y hay que volver a añadirlas.

La tecnología también nos ha dado problemas, por ejemplo, si una clase es hija de la clase MonoBehaviour no podemos hacer new, por lo que para hacer las pruebas nos ha sido necesario modificar algunas de las clases a probar, también hemos necesitado hacer una copia del proyecto para realizar las pruebas ya que al añadirlas la tecnología no reconoce las librerías.

- Player
  - Defecto

Comprobamos que el valor por defecto al crear el objeto es correcto, en este caso comprobamos que se inicializa a 1000 de dinero

  - Aumentar

Comprobamos que la función aumentar funciona correctamente, esta función lo que hace es aumentar el dinero que posee el jugador, y no puede quitar.

  - Pagar

Comprobamos que la función pagar funciona correctamente, esta función devuelve true en el caso de que sea posible pagar y false en el que no, además quitará dinero en el caso de que sea posible.

- Turno
  - SiguienteTurno

Comprobamos que al ejecutar la función siguienteTurno cambiamos de turno.

  - JugadorActual

Comprobamos que la función JugadorActual nos devuelve un jugador, además que al cambiar de turno (ejecutar la función siguienteTurno) que cambiamos de jugador.

Código Player

```
using NUnit.Framework;

namespace Testing
{
	public class UnitTestPlayer
	{
		//HA SIDO NECESARIO HACER QUE PLAYER NO HEREDE DE MonoBehaviour
		Player plr;
		[SetUp()]
		public void Init()	{		plr = new Player(); }

		[Test()]
		public void Defecto()		{		Assert.AreEqual(plr.GetDinero(),1000);		}

		[Test()]
		public void Aumentar()
			{
				plr.Aumentar(500);
				Assert.AreEqual(1500, plr.GetDinero());

				plr.Aumentar(-500);
				Assert.AreEqual(1500, plr.GetDinero());
			}

		[Test()]
		public void Pagar()
			{
				//Player inicializado a 1000 de dinero
				Assert.AreEqual(true, plr.Pagar(500));
				Assert.AreEqual(500, plr.GetDinero());

				//Ahora el player deberia tener 500 de dinero
				Assert.AreEqual(true, plr.Pagar(500));
				Assert.AreEqual(0, plr.GetDinero());

				//Ahora el player deberia tener 0 de dinero
				Assert.AreEqual(false, plr.Pagar(500));
				Assert.AreEqual(0, plr.GetDinero());
			}
	}
}
```

Código Turno

```
using NUnit.Framework;

namespace Testing
{
	[TestFixture()]
	public class UnitTestTurno
	{
		Turno t;
		Player plr1, plr2;

		[SetUp()]
		public void Init()
			{
				t = new Turno();
				plr1 = new Player();
				plr2 = new Player();
			}

		[Test()]
		public void SiguienteTurno ()
			{
				for(int i=1;i<=100;i++)
					{
						Assert.AreEqual(t.GetTurno(), i);
						t.SiguienteTurno();
					}
			}

		[Test()]
		public void JugadorActual ()
			{
				t.AddPlayer(plr1);
				t.AddPlayer(plr2);

				Assert.AreEqual(t.JugadorActual(), plr2);
				t.SiguienteTurno();

				Assert.AreEqual(t.JugadorActual(), plr1);
				t.SiguienteTurno();

				Assert.AreEqual(t.JugadorActual(), plr2);
			}	}	}
```

8- Manual de Usuario

Controles

- **Movimiento del ratón/teclas de dirección**: Movimiento de la cámara.
- **Click izquierdo**: Sobre una unidad, selección de unidad. Click otra vez en el rango de movimiento para moverse. Sobre un botón, confirmación de opción.
- **Click derecho**: Des-selecciona una unidad u objeto. Retrocede en la navegación de menús.
- **Escape:** Abre menú de pausa, que contiene las opciones 'Finalizar turno' y 'Salir del juego'.