Práctica Evaluable Tema 6.

Programación orientada a objetos

Objetivos.

- Repasar los conceptos estudiados hasta ahora.

Consideraciones iniciales.

- La práctica consiste en un único proyecto a realizar en Visual Studio o similar, que se evaluará sobre 10 puntos
- Cada clase debe estar en un fichero fuente propio, con el nombre de la clase y extensión ".cs", como en los ejercicios que habéis hecho durante este tema 6.

Código implementado

Para cada archivo fuente entregado se deberá incluir como comentario en las primeras líneas del archivo el nombre del autor, y una breve descripción de en qué consiste el archivo o clase.

Además, en la clase principal (la que tenga el método Main) se incluirá un listado de todos los apartados, indicando si han sido implementados totalmente, parcialmente o no ha sido realizado.

Por ejemplo:

```
/*
Perez Gomez, Andres
Practica Evaluable Tema 6
Apartado 1 si / no / parcialmente
Apartado 2 si / no / parcialmente
Apartado 3 si / no / parcialmente
...
*/
```

Entrega.

Se debe entregar un archivo comprimido ZIP con el proyecto Visual Studio completo.

- Nombre del archivo: **Apellidos_Nombre_PracT6.zip**

  Por ejemplo, si te llamas Andrés Pérez Gómez el archivo debe llamarse *Perez_Gomez_Andres_PracT6.zip*

Desarrollo.

Ejercicio "Futbol Sala".

Nombre del proyecto: "FutbolSala"

Puntuación máxima: 10 puntos

La práctica de este tema consiste en la realización de un proyecto en Visual Studio (o algún entorno equivalente), en el que se implementen varias clases con sus relaciones entre ellas.

Se pretende simular el desarrollo de un partido de fútbol sala. Se deben definir distintos tipos de jugadores, creando dos equipos de diez jugadores (5 titulares y 5 suplentes) de distintos tipos y, finalmente, simular un partido con jugadas alternas para cada equipo.

1. Estructura de clases para los jugadores

Para gestionar los jugadores, se necesitan estas clases:

1.1 Clase Jugador (1,5 puntos)

Clase *abstracta* llamada **Jugador**, con los datos generales para todos los tipos de jugadores:

- Dorsal del jugador (entre 0 y 15).
- Nombre del jugador.
- Altura en centímetros (entre 120 y 220).
- Porcentaje de defensa (entre 0 y 80).
- Porcentaje de ataque (entre 0 y 80).
- Goles marcados.

Además de estos atributos, se deben añadir los siguientes elementos públicos:

- Propiedades *get/set* para acceder a los elementos anteriores. En estas propiedades se deberá controlar que el valor a asignar sea adecuado (por ejemplo, que no se intente asignar a un jugador el dorsal 120).
- Constructor para especificar todos los parámetros, salvo los goles, que se inicializarán a 0 automáticamente
- Un método virtual *Mostrar* que devolverá una cadena con toda la información (salvo los goles), en este formato (por ejemplo, para el jugador con el número de dorsal 2, llamado Juan Pérez, altura 175 cm, con 75% de defensa y 45% de ataque):

```
2. Juan Pérez, 195cm, 75% defensa, 45% ataque
```

  Se valorará de forma positiva la sustitución de este método *Mostrar* por la sobrecarga del método ToString.

- Un método *MostrarResumen* que devolverá una cadena con el número de camiseta, nombre y goles del jugador. Por ejemplo:

```
2. Juan Pérez: 1 gol
```

1.2. Subclases (1 punto)

De esta clase abstracta heredan tres subclases:

- La clase **Portero (P)**, que añade a los atributos del padre otro que indica la capacidad de parar un tiro a puerta (numérico entre 0 y 10). Redefine el método *Mostrar* o *ToString* del padre, según el caso, para añadir esta característica al final de la cadena devuelta.
- La clase **Defensa (Df)**, que añade a los atributos del padre otros dos que indican la capacidad para robar el balón (numérico entre 0 y 10) y la velocidad (numérico entre 0 y 10). Redefine el método *Mostrar* o *ToString* del padre, según el caso, para añadir estas características al final de la cadena devuelta.
- La clase **Delantero (D)**, que añade a los atributos del padre otros dos que indican la capacidad de marcar un gol (numérico entre 0 y 10) y la velocidad (numérico entre 0 y 10). Redefine el método *Mostrar* o *ToString* del padre, según el caso, para añadir estas características al final de la cadena devuelta.

2. Otras clases necesarias

2.1 Clase Equipo (1,5 puntos)

Un equipo tiene un nombre de equipo (cadena de texto), y está formado por 10 jugadores (array), de forma que el primer y último jugador son porteros, los dos siguientes son defensas, los dos siguientes son delanteros, los dos siguientes son defensas y los dos siguientes son delanteros:

P	Df	Df	D	D	Df	Df	D	D	P

- Define estos dos atributos privados en la clase, junto con sus propiedades get/set públicas.
- Añade un constructor para darle nombre al equipo e inicializar el array con el tamaño adecuado.
- Añade los métodos que consideres oportunos para asignar cada jugador al equipo, garantizando que el jugador ocupa la posición correcta (por ejemplo, que no pueda haber un delantero como primer jugador del equipo).
- Añade un método *Mostrar* que saque por pantalla los datos de cada jugador.
- Añade un método *MostrarEstadisticas* que saque los goles marcados por cada jugador.
- Añade un método *Marcador* que devuelva el total de goles marcados por el equipo (recopilando los goles marcados por cada jugador)

2.2. Clase Partido (3 puntos)

El partido esta compuesto por dos equipos, con sus jugadores. Se debe implementar un constructor para inicializar ambos equipos, y un método Jugada, público, que recibe como parámetro el turno del equipo atacante (1 para el primer equipo, 2 para el segundo equipo).

En cada jugada se elige aleatoriamente un jugador de cada equipo, teniendo en cuenta que:

- No pueden enfrentarse simultáneamente los porteros de cada equipo. Si se produce un enfrentamiento entre porteros se deberá elegir otro jugador aleatoriamente de alguno de los equipos.

La jugada se realiza de la siguiente forma:

- El jugador que defiende intentará recuperar el balón. Para ello su capacidad de defensa global es la suma de sus respectivas capacidades generales y propias:
  - Si es un portero, depende de sus capacidades de defensa y de parar un tiro a puerta.
  - Si es un defensa, depende de sus capacidades de defensa, robo de balón y velocidad.
  - Si es un delantero, depende de sus capacidades de defensa y velocidad.
- El jugador que ataca intentará marcar un gol. Para ello su capacidad de ataque global es la suma de sus respectivas capacidades generales y propias:
  - Si es un portero, depende de su capacidad de ataque.
  - Si es un defensa, depende de sus capacidades de ataque y velocidad.
  - Si es un delantero, depende de sus capacidades de ataque, de marcar gol y velocidad.
- En el enfrentamiento entre jugadores, se compara la capacidad global de ataque de un jugador frente a la capacidad global de defensa del contrincante. Para cada jugador se selecciona su capacidad global de ataque o defensa en función del turno asignado al equipo, ataque o defensa.
  - El atacante podrá tirar a puerta si su capacidad global de ataque es mayor que la capacidad de defensa del contrincante.

    Dependiendo de su porcentaje de ataque global marcará o no. Para ello, se generará un número aleatorio, y si es menor o igual que su porcentaje de ataque global, marcará (y le sumaremos los goles al atacante). De lo contrario, finalizará el ataque sin marcar.
  - El defensor recuperará el balón si su capacidad global de defensa es mayor que la capacidad global de ataque del contrincante. En este caso, la jugada habrá finalizado.

Toda esta lógica debe ser implementada en el método Jugada de esta clase Partido, pero, dada su complejidad, se valorará positivamente que el código se modularice empleando métodos privados, dentro de la propia clase.

3. El programa principal (2 puntos)

El programa principal (clase *Program* que crea Visual Studio por defecto) recibirá dos parámetros por línea de comandos (a través del Main),

- El primer parámetro indica el modo de juego: R para resumido, y C para completo.
- El segundo parámetro indica el número de jugadas, entre 10 y 30 jugadas.
- Si uno o ambos parámetros son incorrectos, finalizará la ejecución del programa mostrando un mensaje por consola indicando el parámetro incorrecto.

Una vez leídos los dos parámetros, si son correctos, se procede a:

- Pedir por consola los datos de los equipos (nombre del equipo y jugadores del mismo).
- Si elige el modo completo (C), se mostrarán por pantalla todas las jugadas del partido, una por línea, y al final el resultado final.

- Si elige el modo resumido (R), se mostrará únicamente el resultado final, y un desglose de goles por jugador para cada equipo. Esta puntuación y desglose también deberá mostrarse en el modo completo anterior.

Un ejemplo de jugadas para los equipos "ElPozo Murcia" y "Viña Albali".

En el modo “completo” se mostraría una salida por pantalla similar a la siguiente:

```
Ataca ElPozo Murcia. Juega Juan López. Tira a puerta. Gol!
Ataca Viña Albali. Juega Sergio Lozano. Roba Mario Martínez.
Ataca ElPozo Murcia. Juega Rafael García. Tira a puerta. Falla!
...
```

En el modo “resumido” se mostraría una salida por pantalla similar a la siguiente:

```
ElPozo Murcia 6 – 4  Viña Albali

Estadísticas ElPozo Murcia:
Rafael García (Defensa): 1 gol
Juan López (Delantero): 3 goles
Mario Martínez (Delantero): 0 goles
…
Estadísticas Viña Albali:
José Pérez (Portero): 0 goles
Sergio Lozano (Defensa): 1 gol
Luis Suárez (Delantero): 3 goles
…
```

4 (1 punto) Además de la implementación del programa siguiendo los pasos indicados, se deberá tener en cuenta:

- La usabilidad, es decir, que el usuario que lo emplee sepa en todo momento lo que tiene que introducir, y se le muestre la información adecuada.
- La utilización adecuada de métodos y funciones auxiliares.
- La no repetición de código innecesario.
- Para mostrar datos por pantalla o pedir datos al usuario solo estará permitido utilizarlas siguientes funciones de Console: ReadLine, Read, WriteLine, Write, Clear.
- En todas las peticiones de datos por consola, en el caso de que algún dato sea incorrecto, se pedirá continuamente hasta que sea correcto. Se penalizará negativamente que se produzca un error en tiempo de ejecución por la introducción de datos incorrectos.

Observaciones generales de la práctica

- Se deberá controlar con TryParse o mediante un bloque *try-catch-finally* que cualquier dato numérico (entero o real) que se pida sea correcto, y esté entre los límites acordados según el dato en cuestión.
- Cualquier dato de tipo texto será válido siempre que no esté vacío. Si está vacío, se volverá a pedir de nuevo.
- En cada clase que se defina (salvo el programa principal), es OBLIGATORIO que los atributos sean privados o protegidos (esto último en caso de que la clase tenga subclases). Se deberá definir en cada una de estas clases, al menos, un **constructor** que reciba tantos parámetros como atributos tenga la clase, y asigne cada parámetro a su atributo correspondiente. Además, se deben definir las correspondientes **propiedades get/set** públicas para acceder a cada elemento privado.
- Se deberá también redefinir (*override*) el método *ToString* en las clases de las que se quiera sacar algo de información por pantalla, para indicar en dicho método el formato de salida de la información.
- Además de los elementos a añadir indicados en cada clase, puedes añadir otros atributos, constructores o métodos si lo consideras oportuno. Se valorará después si esos elementos añadidos son de utilidad o no.
- Se valorará negativamente:
  - La repetición innecesaria de código en cualquier parte del programa. Por ejemplo, volver a asignar manualmente en el constructor de una clase hija atributos que ya asigna la clase padre, o repetir el código para pedir al usuario los datos del transporte de una etapa, dependiendo del tipo de transporte
  - Las funciones o métodos excesivamente largos o complejos, como por ejemplo, y sobre todo, el programa principal, que deberá dividir su funcionalidad adecuadamente en funciones auxiliares.
- Recuerda que se valorará negativamente la suciedad de código, en lo referente a lo visto en el Tema 9 del bloque 1 del módulo de Entornos de Desarrollo. Fundamentalmente, evaluaremos negativamente las siguientes malas prácticas:
  - Nombres poco apropiados de variables, funciones o clases (ya se venía penalizando con anterioridad).
  - Espaciado y alineación vertical (separación entre funciones y entre bloques de código con propósito diferente).
  - Espaciado y alineación horizontal (incluyendo que las líneas de código no excedan del ancho recomendado).
- Recuerda también que, en las funciones o métodos que devuelvan algún tipo de dato (*return*), se valorará negativamente que haya más de un punto de salida. Las buenas prácticas de programación indican que las funciones deben tener un único punto de salida (una única instrucción *return*).
- También se valorará negativamente la utilización incorrecta de las estructuras de control, estructuras repetitivas, las instrucciones,... Algunos casos típicos que :
  - Utilización incorrecta de la estructura de control “switch-case”.
  - La instrucción “goto” no debe utilizarse salvo en “switch-case”.
  - Los bucles “for” sólo deberían utilizarse cuando se conoce el principio y fin del bucle.