Universidad Francisco de Vitoria
Escuela Politécnica Superior
Programación Orientada a Objetos
Profesor: Susana Bautista Blasco

Hundir la flota

—

Práctica POO

2º Cuatrimestre 2022-2023

Realizado por:
Luis Eduardo de Santiago Martínez
Carmen Moreno Udaondo
Oscar Sánchez Vidal

Grado Ingeniería Informática, 1ºC

Índice de contenidos

Resumen

Hundir la flota (o Battleship) es un juego de estrategia para dos jugadores. Se utiliza un tablero cuadriculado en el cual se colocan las flotas de barcos de los diferentes jugadores. La localización de estas flotas se encuentra oculta al otro jugador.

Los jugadores se turnan para "bombardear" a las flotas del otro jugador, siendo el objetivo del juego acabar con todos los barcos del oponente.

El objetivo principal del proyecto consistía en crear el juego de Batalla Naval para consolas. Para llevar a cabo la práctica era necesario implementar diferentes conceptos de la programación orientada a objetos como el encapsulamiento, la herencia, composición y abstracción. Este documento da un breve análisis del desarrollo de nuestra solución en C# explicando las diferentes decisiones e implementaciones realizadas.

Luis Eduardo de Santiago Martínez

Carmen Moreno Udaondo

Oscar Sánchez Vidal

Descripción general

La práctica realizada pretende crear una aplicación de consola que reproduce el juego de Battleship ofreciendo dos modalidades de juego:

1. Partidas jugador humano VS jugador humano.
2. Partidas jugador humano VS jugador automático.

La propia aplicación ofrece diferentes funcionalidades propias de un juego:

- **Cargar partida anterior**: Muestra un listado de las diferentes partidas que han sido guardadas y no se encuentran finalizadas. El jugador podrá elegir la partida a continuar y esta será cargada desde el punto de juego en el que se encontraba la partida.
- **Partida de 2 jugadores**: Crear una nueva partida desde cero donde ambos jugadores son humanos.
- **Partida de 1 jugador**: Crear una nueva partida desde cero donde uno de los jugadores será humano y el otro automático.
- **Ranking**: Muestra un listado de todas las partidas que han finalizado, especificando los jugadores, el número de movimientos y el ganador de la partida.
- **Salir**: Esta funcionalidad finaliza la aplicación actualizando los ficheros de *PartidasFinalizadas.bin* y *PartidasGuardadas.bin*.

Las partidas están formadas por tres fases:

1) **Configuración**: Inicio de la partida donde los jugadores introducirán sus nombres y la localización de sus respectivas embarcaciones. En el caso de que uno de los jugadores sea automático, se le asignará el nombre "Autómata" y se colocarán sus barcos de forma aleatoria.
2) **Juego**: Secuencia de turnos en la que los jugadores atacan los mapas del oponente.
3) **Finalización**: El fin da la partida puede producirse de dos formas: Mediante la victoria de un jugador tras hundir el último barco del contrincante o tras salir de la partida antes de realizar un disparo.

El mapa que utilizan los jugadores se trata de una cuadrícula de 12x12 casillas. Ciertas casillas corresponden a zonas de tierra, las cuales se encuentran en la misma posición para todas las partidas.

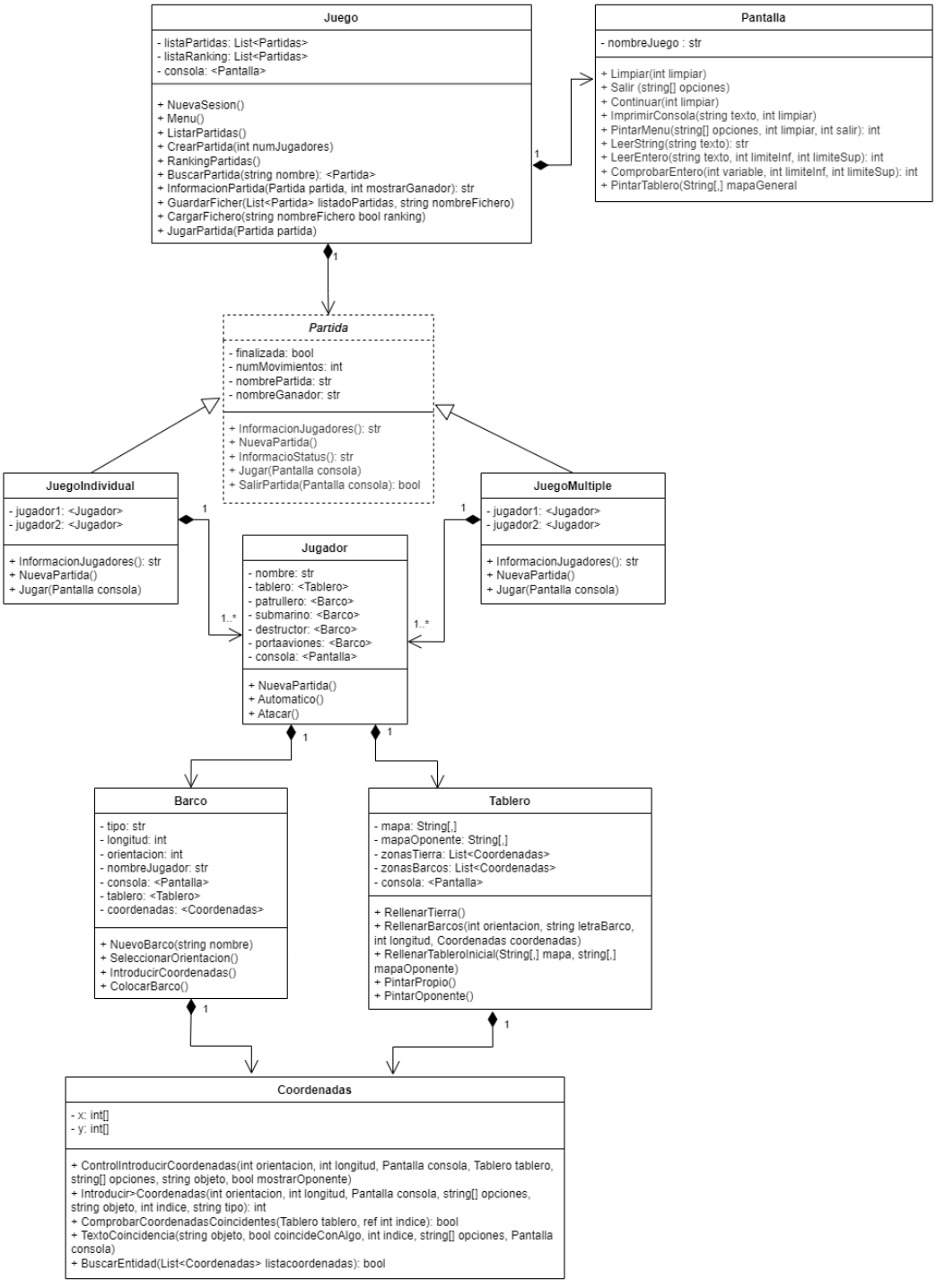
Está decisión la tomamos ya que creímos que una vez las zonas se colocasen correctamente, debíamos priorizar otras partes del desarrollo, como la posibilidad de guardar partidas, antes de aleatorizar la posición de las zonas de tierra.

Control de versiones

Previamente a comenzar a pensar y picar código, decidimos hablar y elegimos que plataforma utilizaríamos para el control de versiones y la colaboración. Al final nos decantamos por Github dado que algunos de nosotros habíamos trabajado con esta plataforma anteriormente.

Clases y sus métodos

En primer lugar, diseñamos un diagrama de clases que describía la estructura estática de nuestra práctica. El diagrama final que obtuvimos fue el resultado de varios replanteamientos dado que como describiremos en posteriores apartados, tuvimos que enfrentarnos a varios problemas derivados de nuestro primer diseño.



Nuestra solución cuenta con un total de 9 clases. A grandes rasgos, contamos con 1 clase abstracta y una profundidad de herencia (mide la longitud de la jerarquía de herencia de los objetos) de 2.

En primer lugar, encontramos la clase Juego que define los atributos y métodos necesarios para el correcto funcionamiento de la gestión de partidas de nuestro juego. Decidimos crear una clase a parte ya que queríamos seguir la buena práctica de reducir la complejidad ciclomática en el *Main*.

Los atributos de Juego son dos listas de instancias de la clase Partidas que almacenan todas las partidas de nuestro juego y las partidas finalizadas. Utilizamos listas en vez de arrays ya que, en C# al crear un array, se debe especificar su tamaño. Por otro lado, contamos con un atributo llamado consola. Este se trata de una instancia de la clase Pantalla, la cuál utilizamos para controlar la entrada y salida de datos del usuario.

En cuanto a los métodos de esta clase, todos cuentan con nombres descriptivos que explican su funcionalidad: Menú, BuscarPartida, RankingPartidas, CrearPartidas, etc.

La clase Partida es una clase abstracta la cual sirve de *plantilla* para sus clases hijas que son PartidaMultiple y PartidaIndividual. Mediante la clase Partida controlamos las partidas del juego. Dicho de otra forma, la creación, el estado, la información de los jugadores y la dinámica de juego de las partidas son controlados desde estas clases. Para ello utilizamos el polimorfismo mediante la implementación de métodos virtuales en la clase base, a los cuales proporcionamos su propia definición en la clase hija/derivada.

Dentro de PartidaMultiple y PartidaIndividual contamos con dos instancias de la clase Jugador que representan al jugador humano 1, jugador humano 2 y al jugador automático en sus respectivas partidas.

A la hora de diseñar los jugadores planteamos varias opciones. Contemplamos la posibilidad de tener clases hijas de Jugador que representasen y controlase las funcionalidades de los jugadores humanos y automáticos.

Finalmente nos decantamos por tener una sola clase de jugador que controla y verifica las diferentes acciones de los jugadores.

Cada jugador cuenta con una serie de atributos que incluyen su nombre, para poder identificarlo en todo momento, diferentes instancias de la clase Barco que representan los diferentes navíos de cada jugador y el tablero.

A la hora de implementar Barco solo creamos una clase ya que la característica que varía dependiendo del navío es la medida de la longitud.

Por último, en cuanto al tablero, nos decantamos por que el mapa fuese un array de strings dado que nos parecía la forma más asequible de abordar el editar las casillas. Para ello, creamos la clase Coordenadas mediante la cual insertamos nuevas coordenadas, atacamos a otras y podemos colocar embarcaciones, zonas de tierra, etc.

La funcionalidad y definición de todos los atributos se encuentra comentado dentro del propio código. Asimismo, hemos utilizado etiquetas xml para poder documentar nuestra solución.

Clase Pantalla

Creemos que es importante profundizar más en nuestra clase Pantalla. Esta clase controla todas las operaciones de entrada y salida de datos del programa. De esta forma, su propósito es aislar la interfaz del resto de la solución.

Dado que en cada clase solo deben de estar los métodos relativos a la clase, imprimir en pantalla, por tanto, no es algo propio de la misma.

Así fue como nos propusimos el reto de seguir está buena práctica y utilizar la clase Pantalla. Un claro ejemplo de esto es la función PintarMenu. Se trata de un método al que le pasamos un string array que representa las diferentes opciones del menú, un entero que representa si deseamos limpiar la consola antes de mostrar el texto y otro entero que representa si se desea permitir la opción de salir en el menú.

No solo mostramos texto por consola, si no que también lo recogemos. El método LeerEntero muestra por consola un texto y lee el input del usuario. Recibe tres parámetros: un string que representa el texto que se quiere mostrar, un entero que representa el límite inferior del intervalo de número que puede introducir el usuario y un entero que representa el límite superior.

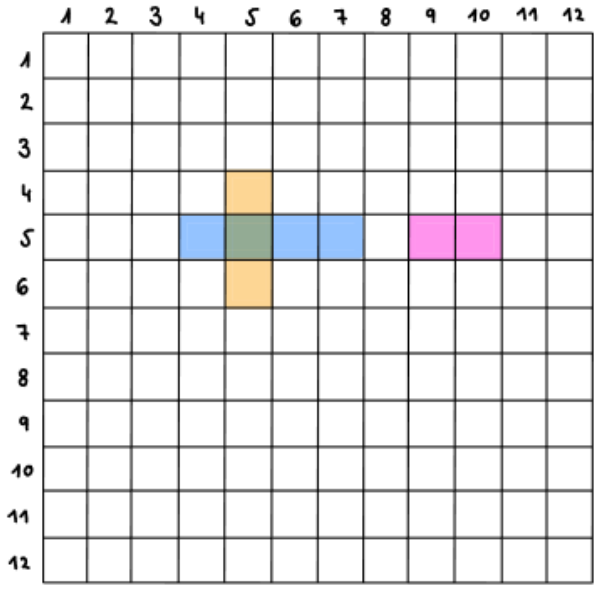
Dado que en el enunciado de la práctica especificaba que se considera que todos los datos introducidos por el usuario son correctos, no comprobamos mediante un *try catch* si los strings se pueden convertir a enteros.

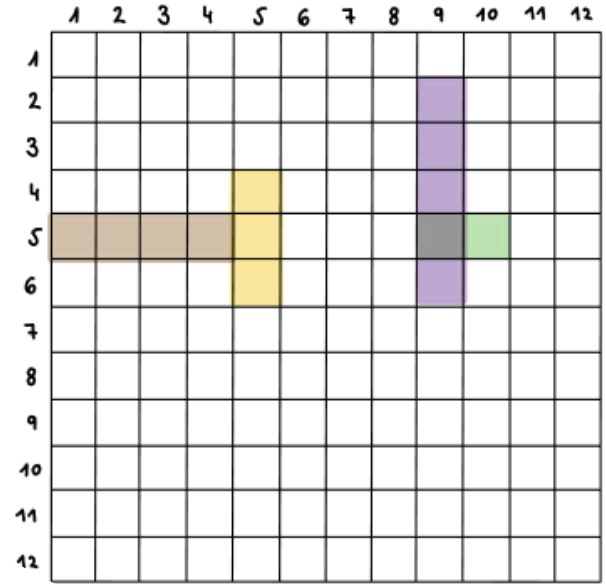
Comprobar coincidencia

Debido a la complejidad de la instrucción if que utilizamos para la coincidencia de objetos en las coordenadas introducidas (método BucarEntidad en la clase Coordenadas). Vemos oportuno explicar el proceso que realizamos para llegar hasta ella.

Dado que las condiciones que usábamos no funcionaban, decidimos recurrir al método infalible del lápiz y papel. Por ello estuvimos comprobando a mano las

diferentes combinaciones que resultaban erróneas y el proceso lógico mediante el cual nosotros comprobábamos la coincidencia.






```
for (int i = 0 , i < listaCoords.Count; i++)
{
  if (coord.x[0] <= listaCoords[i].x[0] <= coord.X[1])
  {
    if( listaCoords[i].y[0] <= coord.y[0] <= listaCoords.y[i].y[1])
    {
      true
    }
  }
}
```


```
for (int i = 0 , i < listaCoords.Count; i++)
{
  if (listaCoords[i].x[0] <= coord.x[0] <= listaCoords[i].x[1])
  {
    if (coord.y[0] <= listaCoords[i].yo <= coord.y[1])
    {
      true
    }
  }
}
```

Ficheros y su formato

A la hora de implementar el almacenamiento de datos en ficheros fuimos probando muchas opciones, pero finalmente nos quedamos con los ficheros binarios.

Nuestra clase Juego cuenta con dos atributos que son Listas de las diferentes partidas. ListaPartidas almacena todas las partidas creadas, mientras que listaRanking solamente almacena aquellas partidas que han finalizado.

La primera vez que abordamos el almacenamiento en ficheros intentamos serializar estas dos listas en un fichero .txt y, aunque esto funcionaba, nos encontramos con dificultades para deserializar los datos almacenados.

Posteriormente probamos a guardar estas dos listas en archivos .JSON dado que tienen una estructura muy ordenada y facilitaban mucho la serialización y deserialización de las listas. El problema que nos encontramos con este enfoque fue nuestra clase abstracta. Esto se debió a que el paquete/libre NuGet System.Txt.Json que se utiliza para almacenar datos en ficheros .JSON tiene problemas para serializar clases abstractas. Aunque nosotros no creamos directamente instancias de la clase Partida, las listas listaPartidas y listaRanking se tratan de listas polimórficas de la clase Partida.

Por ello y tras bastante investigación nos decantamos por los archivos binarios. Para ello utilizamos el *BinaryFormatter* que serializa y deserializar un objeto, o todo un grafo de objetos conectados, en formato binario.

Dentro de la clase Juego contamos con dos métodos, GuardarFichero y CargarFichero, que se encargan de guardan una lista de forma síncrona en su respectivo fichero y cargar un objeto de un fichero.

Cargamos los datos almacenados en los ficheros nada más arrancar el programa. Por otro lado, guardamos los datos pertinentes en los ficheros al salir mediante la opción 5 del menú, al crear una nueva partida y al acabar la fase de finalización de las partidas. A la hora de cargar ficheros, si estos no existen se crean para posteriormente almacenar los datos necesarios.

Utilizamos rutas relativas para almacenar los datos. Estas rutas las representamos mediante dos constantes de tipo string declaradas en *Program.cs* .

Conclusiones

Gracias al desarrollo de esta práctica hemos podido trabajar y observar de primera mano los conceptos de la programación orientada a objetos impartidos en clase como son la abstracción, el polimorfismo, la herencia, etc. Asimismo, pudimos trabajar el almacenamiento y lectura de datos a un fichero, trabajando e investigando las diferentes formas en las que se puede hacer.

En cuanto al trabajo en equipo, hemos tenido que aprender a dividir correctamente el trabajo ha realizar. Tres personas trabajando en el código si no están organizadas es imposible que logren sacar el código a delante.

Particularmente decidimos aprovechar la función de proyectos que ofrece Github para dividir las tareas, asignarlas y realizar un seguimiento de estas.

Porcentajes de trabajo:

- Luis Eduardo de Santiago Martínez -> 35 %
- Carmen Moreno Udaondo -> 40 %
- Oscar Sánchez Vidal -> 25 %

Referencias

Barcia Santos, I., & Bautista Blasco, S. (s.f.). Práctica 2 - Hundir la flota. Obtenido de Ufv Canvas: https://ufv-es.instructure.com/courses/26481/pages/practica-2-hundir-la-flota?module_item_id=632234

BillWagner, d. G. (s.f.). Polimorfismo. Obtenido de Microsoft Learn: https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/object-oriented/polymorphism

BinaryFormatter Class. (s.f.). Obtenido de Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.binary.binaryformatter?view=net-8.0

Diagramas del UML - Madrid. (s.f.). Obtenido de https://www.teatroabadia.com/es/uploads/documentos/iagramas_del_uml.pdf

Wikipedia Foundation. (s.f.). Battleship (game). Obtenido de Wikipedia: https://en.wikipedia.org/wiki/Battleship_(game)