Fundamentos de la programación

Práctica 3. Dungeons[1]

Indicaciones generales:

- Las líneas 1 y 2 deben ser comentarios de la forma `// Nombre Apellido1 Apellido2` con los nombres de los alumnos autores de la práctica, en todos los archivos presentados.
- **Lee atentamente** el enunciado y realiza el programa tal como se pide, sin cambiar la representación dada, implementando las clases y métodos que se especifican, y respetando los parámetros y tipos de los mismos. Pueden implementarse los métodos auxiliares que se consideren oportunos comentando su cometido, parámetros, etc.
- La entrega se realizará a través del campus virtual, subiendo únicamente un .zip con los archivos fuente `Listas.cs`, `Map.cs`, `Dungeon.cs` y `Program.cs` (deben respetarse estos nombres para facilitar la corrección).
- El programa, además de ser correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.
- El **plazo de entrega** finaliza el 8 de mayo.

Las aventuras conversacionales son un género de juegos de ordenador (también conocido como *Interactive Fiction*[2]) que se caracteriza por establecer un diálogo con el jugador en el que la máquina presenta descripciones (principalmente textuales) de lo que ocurre en el juego y el jugador responde escribiendo órdenes para indicar las acciones que desea realizar.

El **motor** de una aventura conversacional es habitualmente un intérprete que recibe como entrada un archivo con la especificación de una aventura concreta, que incluye datos y descripciones narrativas (a menudo impregnadas de cierto valor literario) sobre las localizaciones del mundo del juego, los objetos y personajes se encuentran allí, etc. El motor procesa dicha especificación creando una experiencia interactiva para que los jugadores se sitúen en ese entorno virtual, lo exploren y traten de superar con éxito los diversos retos que presenta el juego. Para ello el jugador tiene a su disposición un repertorio de acciones posibles a realizar, como moverse de una localización a otra, examinar y usar objetos, interactuar con otros personajes, etc.

En esta práctica vamos a implementar una versión simplificada de la aventura conversacional *Dungeon*[3]. El juego consiste controlar las acciones del jugador utilizando un pequeño repertorio de instrucciones para moverlo en un mapa. El mapa consta de un conjunto de lugares conectados por puertas que pueden estar situadas en las 4 direcciones básicas (norte, sur, este y oeste). Cada lugar tiene un *nombre* y una *descripción* con información sobre lo que hay en el mismo. Algunos lugares representan la salida del mapa, donde termina la aventura. Los lugares también pueden contener enemigos que pueden atacar al jugador, así como ser atacados por el mismo. Cuando el jugador llega a un lugar se muestra en pantalla su descripción, las puertas que tiene y los enemigos que hay. El objetivo del juego es alcanzar la salida en el mapa antes de que el jugador muera.

El jugador y los enemigos se caracterizan por dos atributos: puntos de salud (*Health Points* o HP) y puntos de ataque (*Attack Points* o ATK). Cada vez que el jugador entra en un lugar es atacado por todos los enemigos que hay en él, es decir, los HP del jugador se reducen en una cantidad que es la suma de los ATK de todos los enemigos que hay en el lugar.

---

[1] Esta práctica ha sido diseñada en colaboración con Guillermo Jiménez Díaz.

[2] Para saber algo más: `http://en.wikipedia.org/wiki/Interactive_fiction`

[3] Basándonos en una práctica propuesta en la asignatura de Tecnología de la Programación por los profesores P. Arenas, M.A. Gómez, J. Gómez y G. Jiménez.

El jugador puede atacar a los enemigos que hay en el lugar. Cada vez que los ataca, inflige un daño ATK a cada uno de ellos. Después, todos los enemigos devuelven el ataque. El jugador puede atacar reiteradamente a los enemigos, recibiendo a su vez el ataque correspondiente.

Los comandos que entiende el jugador son los siguientes:

- `go` seguido de una dirección (`north`, `south`, etc): se mueve en la dirección indicada
- `enemies`: muestra los enemigos que hay en el lugar actual.
- `attack`: ataca a los enemigos que hay en el lugar actual.
- `info`: muestra la información del lugar actual.
- `stats`: muestra las estadísticas (HP y ATK) del jugador.
- `quit`: termina el juego a petición del jugador.

Los mapas se leen de un archivo con el siguiente formato (ver mapa de ejemplo "hauntedHouse.map"):

```
...
dungeon 3 BlueBedroom noExit
"The big bed occupies half the room.
 Father died in this place"
...

dungeon 9 Garden exit
"That small door led you to the garden.
Finally you can breathe"
...

door 0 dungeon 0 north dungeon 3
door 1 dungeon 0 south dungeon 9
...

enemy 0 Ghost 5 1 dungeon 0
enemy 1 Witch 2 1 dungeon 0
...
```

Un mapa contiene ítems de 3 tipos, que se identifica por la primera palabra. Veamos un ejemplo de cada tipo de ítem:

- `dungeon 3 BlueBedroom noExit`: define el lugar 3, que se llama `BlueBedroom` y no es salida (`noExit`). Todo el texto que aparece entrecomillado en las siguientes líneas (`"The big bed occupies..."`) corresponde a la descripción del lugar. En este mapa el lugar 9 correspondería a una salida.
- `door 0 dungeon 0 north dungeon 3`: define la puerta 0, que conecta el lugar 0 al norte, con el lugar 3. Las puertas pueden conectar en las 4 direcciones habituales: `north`, `south`, `east`, `west`, y son bidireccionales, es decir, la puerta del ejemplo también establece una conexión entre el lugar 3 y el lugar 0 en dirección `south`.
- `enemy 0 Ghost 5 1 dungeon 0`: define el enemigo 0 con nombre `Enemy0`, con valores `HP` 5 y `ATK` 1, en el lugar 0.

Los lugares aparecerán en el archivo numerados en orden creciente, desde el 0 en adelante, así como las puertas y los enemigos. Asumimos que el mapa es correcto, es decir, que las puertas conectan lugares existentes, los enemigos aparecen en lugares también existentes, etc. Además los enemigos ubicados en los lugares son únicos, es decir, dos lugares no pueden contener el mismo enemigo.

Para implementar el juego vamos a utilizar la clase `Listas` de enteros vista en clase, **extendiéndola con los métodos que sean necesarios**. Además definiremos otras clases más, todas ellas en el espacio de nombres `Dungeon`. La primera es la clase `Map`, para la carga y manejo de los mapas:

```
namespace Dungeon{
 // posibles direcciones
 public enum Direction {North,South,East,West};

 class Map{
  struct Enemy{
    public string name, description;
    public int attHP, attATK;
  }

  // lugares del mapa
  struct Dungeon {
    public string name, description;
    public bool exit;    // es salida?
    public int [] doors; // vector de 4 componentes con el lugar
                         // al norte, sur, este y oeste respectivamente
                         // -1 si no hay conexion
    public Lista enemiesInDungeon; // lista de enteros, indices al vector de enemigos
  }

  Dungeons [] dungeons;    // vector de lugares del mapa
  Enemy [] enemies;        // vector de enemigos del juego
  int nDungeons, nEnemies; // numero de lugares y numero de enemigos del mapa
```

Los enemigos y los lugares del mapa vienen definidos por el nombre y la descripción. Todos los enemigos del mapa se guardan en el vector `enemies` (el enemigo 0 en la componente 0, el 1 en la componente 1, etc.) y los lugares se guardan en el vector `dungeons`. Los lugares contienen además otra información:

- Si el lugar es o no salida (campo `exit`).
- Las puertas de ese lugar se guardan en un vector `doors`. La componente 0 contiene **el numero del lugar** (referente al vector `dungeons`) con el que está conectada al norte; la componente 1, el lugar al sur, etc. En caso de no haber puerta en una dirección dada, esa componente valdrá -1.
- Los enemigos que hay en el lugar, que se guardan en una lista `enemiesInDungeon` de índices que se refieren al vector `enemies`.

Antes de continuar conviene hacer un **dibujo de esta estructura** y comprender bien cómo se almacena la información de un mapa. Los métodos a implementar para esta clase son:

- `public Map(int numDungeons, int numEnem)`: constructora de la clase. Genera un mapa vacío con el número de lugares y de enemigos indicados.
- `public void ReadMap(string file)`: lee el mapa del archivo `file` y lo almacena en la estructura descrita anteriormente. Para facilitar la lectura implementaremos tres métodos

privados auxiliares: `CreateDungeon`, `CreateDoor`, `CreateEnemy`. Una vez leída la primera línea del ítem y determinado su tipo, se llamará al método correspondiente que creará el ítem en cuestión (utilizando los parámetros adecuados).

Para facilitar la lectura, también será útil un método para leer la descripción de los lugares (`private string ReadDescription(StreamReader f)`), que tendrá en cuenta que dicha descripción puede aparecer en varias líneas del archivo de entrada (la descripción va delimitada por comillas, y puede contener saltos de línea).

- `public string GetDungeonInfo(int dung)`: devuelve toda la información del lugar indicado `dung`. Por ejemplo, para el dungeon 0, devolverá la cadena (incluidos saltos de línea):

```
Dungeon: BallRoom
You are at the ancient BallRoom, where all that people died
Note that there is a big clock and a place for the orchestra
Exit: False
```

- `public string GetMoves(int dung)`: devuelve los movimientos posibles desde el lugar `dung`. Por ejemplo, para el lugar 1 (véase el mapa "hauntedHouse.map"), devolverá:

```
north: Garden
east: BallRoom
```

- `public int GetNumEnemies(int dung)`: devuelve el número de enemigos que hay en el lugar `dung`.
- `private string GetEnemyInfo(int en)`: devuelve la información sobre el enemigo `en` de la lista de enemigos. Para el enemigo 0 devuelve:

```
0: Enemy0  "Bad guy" HP 5 ATK 1
```

- `public string GetEnemiesInfo(int dung)`: devuelve la información sobre todos los enemigos que hay en el lugar `dung`. Para el lugar 0 devuelve:

```
0: Enemy0  "Bad guy"  HP 5   ATK 1
1: Enemy1  "Worse guy"  HP 2   ATK 1
```

- `public int GetEnemyATK(int en)`: devuelve el valor `attATK` del enemigo `en`.
- `private bool MakeDamageEnemy(int en, int damage)`: resta `damage` unidades al atributo `HP` del enemigo `en`, si existe el mismo. Si ese atributo `HP` es ≤ 0 devuelve *true* (enemigo muerto), *false* en caso contrario.
- `public int AttackEnemiesInDungeon(int dung, int damage)`: aplica el daño `damage` correspondiente a cada uno de los enemigos que hay en el lugar `dung`. Los enemigos muertos son eliminados de la lista correspondiente. Devuelve el número de enemigos que han sido eliminados tras el ataque.
- `public int ComputeDungeonDamage(int dung)`: devuelve la suma de los valores de ATK de todos los enemigos que hay en el lugar `dung`.
- `public int Move(int pl, Direction dir)`: devuelve el lugar al que se llega desde el lugar `pl` avanzando en la dirección `dir` (-1 en caso de error).

- `public bool isExit(int dung)`: comprueba si el lugar `dung` es salida del mapa.

A continuación implementaremos la clase `Player` para representar el estado del jugador. Esta clase tendrá únicamente tres atributos para determinar su posición y sus valores `health` y `damage`:

```
namespace Dungeon {
    class Player {
        int pos;   // posicion del jugador en el mapa
        int health, damage;
```

Implementará los siguientes métodos:

- `public Player()`: constructora de la clase, que sitúa al jugador en la posición 0 y con valores 10 y 2 para `HP` y `ATK`. Estos valores deberán declararse como constantes para poder modificarlas fácilmente.
- `public int GetPosition()`: devuelve la posición actual del jugador.
- `public bool IsAlive()`: devuelve *true* si el valor `HP` es mayor que 0; *false* en caso contrario.
- `public string PrintStats()`: devuelve una cadena con las estadísticas del jugador:

  ```
  Player: HP 5 ATK 1
  ```

- `public string GetATK()`: devuelve el valor `ATK`.
- `public bool ReceiveDamage(int damage)`: reduce el HP del jugador de acuerdo al valor *damage*. Devuelve *true* si el jugador sigue vivo tras recibir el daño; *false* en caso contrario.
- `public void Move(Map m, Direction dir)`: mueve el jugador en la dirección `dir` a partir de la posición actual de acuerdo con el `Mapa m` (utiliza el método `Move` de la clase `Map`).
- `public bool atExit(Map map)`: indica si jugador está en una salida del mapa `map`.

Por último, la clase `Main`, también dentro del *namespace* `Dungeon`, hace uso de las clases anteriores e implementará los métodos:

- `static void ProcesaInput(string com, Player p, Map m)`: procesa el comando representado en la cadena `com`.
- `bool EnemiesAttackPlayer(Map m, Player p)`: los enemigos que están en el lugar en el que encuentra el jugador le atacan. Devuelve `true` si hay enemigos que ataquen al jugador; `false` en caso contrario.
- `int PlayerAttackEnemies(Map m, Player p)`: el jugador ataca a los enemigos que se encuentran en el lugar. Este método devuelve el número de enemigos que han sido eliminados.
- `Main`: crea un mapa, lo lee del archivo dado e implementa el bucle principal del juego. En cada iteración se escribe en pantalla el prompt ">", se lee un comando para el jugador y se procesa con el método anterior hasta alcanzar un punto de salida o hasta que el jugador aborte el juego (comando `quit`) o el jugador muera.

Una vez implementadas las clases descritas, se pide implementar las funcionalidades siguientes:

- Utilizar excepciones para hacer un tratamiento adecuado de los casos inesperados en los métodos definidos.
- Implementar un método que permita leer de archivo una lista de comandos para el jugador (un comando por línea) y haga que jugador ejecute dichar instrucciones en secuencia.
- Extender la implementación para poder interrumpir una partida y recuperarla posteriormente. Debe guardarse el estado actual del juego en un formato adecuado, para poder restaurarlo posteriormente.