Fundamentos de la programación

Práctica 3. Adventure Game[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 `Lista.cs`, `Map.cs`, `Player.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 7 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 con una buena dosis de estilo 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 que puede 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 adaptada de `https://github.com/llpk79/AdventureGame`. 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* y en algunos lugares hay objetos o *items* que el usuario puede coger y utilizar más tarde. Estos objetos tienen un peso y pueden proporcionar unos puntos de salud o *health points* (hp). Hay un lugar en el mapa desde el que parte el jugador y otro de salida donde termina la aventura. El jugador se puede mover de un lugar a otro, consumiendo puntos de salud. El objetivo del juego es alcanzar la salida antes de que el jugador muera (es decir, se quede sin puntos de salud).

Los comandos que entiende el jugador son los siguientes:

- `go <direccion>`: se mueve en la dirección indicada (`n`, `s`, `e`, `w`).
- `pick <item>`: recoge el objeto indicado del lugar actual.

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

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

- `look`: muestra los objetos que hay en lugar actual.
- `info`: muestra la información sobre el lugar actual y las direcciones hacia las que se puede mover.
- `inventory`: muestra información sobre el inventario del jugador.
- `eat <item>`: el jugador come el objeto indicado del inventario, eliminando peso y obteniendo los puntos de salud asociados al mismo.
- `me`: nombre del jugador, puntos de salud y peso que lleva.
- `quit`: termina el juego a petición del jugador.
- `help`: muestra el repertorio de comandos y su descripción.

El mapa del juego se lee de un archivo con el siguiente formato (ver mapa de ejemplo "mapa.dat"):

```
room rest_stop "You are at a North Cascades Hwy rest-stop. It is a crisp fall
morning. North of you, a little dog barks and dashes down a forest trail. To the
South, your warm car and the open road."

room car "You head down the road, wondering what could have been. An 18 wheeler
crushes you. Look both ways before entering traffic."
...
conn rest_stop s car
conn rest_stop n forest_trail
...
item flashlight 5 0 rest_stop "A handy tool to light your way."

item black_key 1 0 dense_forest "A black key. Must go to something."
...
entry car
exit doggo
```

Tal como se muestra, el mapa contiene 5 tipos de datos, que se identifican por la primera palabra:

- `room rest_stop "You are ..."`: define el lugar que se llama `rest_stop`. Todo el texto que aparece entrecomillado en las siguientes líneas (`"You are at a North Cascades..."`) corresponde a la descripción del lugar.
- `conn rest_stop s car`: especifica que al sur de `rest_stop` está el lugar `car`. Esta conexión es bidireccional (al norte de `car` está `rest_stop`), pero el archivo solo tendrá explícitamente una de las dos.
- `flashlight 5 0 rest_stop "A handy ..."`: indica que el objeto `flashlight` pesa 5 unidades, proporciona 0 puntos de salud (hp) y está en el lugar `rest_stop`.
- `entry car` y `exit doggo` indican, respectivamente, los lugares de entrada y de salida del mapa.

Asumimos que el mapa es correcto, es decir, que las conexiones se refieren a lugares existentes, los objetos aparecen en lugares también existentes, hay un punto de entrada y al menos otro de salida, etc. Además los objetos ubicados en los lugares son únicos.

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

```
namespace Adventure{
  public enum Direction { North, South, East, West };
  class Map {
      // items
      public struct Item {
          public string name, description;
          public int hp;     // health points
          public int weight; // peso del item
      }
      // lugares del mapa
      public struct Room {
          public string name, description;
          public bool exit;  // es salida?
          public int[] connections; // vector de 4 componentes
                                    // con el lugar al norte, sur, este y oeste
                                    // -1 si no hay conexion
          public Lista itemsInRoom; // indices al vector de items n los items del lugar
      }
      Room [] rooms;       // vector de lugares del mapa
      Item [] items;       // vector de items del juego
      int nRooms, nItems;  // numero de lugares y numero de items
      int entryRoom;       // numero de la habitacion de entrada (leida del mapa)
```

Para cada objeto se guarda en un struct `Item` con toda información referente al mismo; el vector `items` almacena el conjunto de objetos del mapa. De manera análoga, para cada lugar se guarda en un struct `Room` con toda la información leída del mapa; el conjunto de lugares se guarda en el vector `rooms`. Además, para cada lugar (`Room`) se tiene la lista la lista `itemsInRoom` con los (índices a los) objetos de la habitación (el objeto 0 en la componente 0, el 1 en la componente 1, etc.). Las conexiones se guardan en un vector `connections`: la componente 0 contiene **el numero del lugar** (referente al vector `rooms`, que se define más abajo) 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.

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 `Map` son:

- `public Map(int numRooms, int numItems)`: constructora de la clase. Genera un mapa vacío con el número de lugares y de objetos dados en los argumentos.

- `public void ReadMap(string file)`: lee el mapa del archivo `file` y lo almacena en las estructuras descritas arriba. Nótese que hay líneas de comentario que comienzan con `"//"` que directamente no se procesan.

  Para el resto de líneas determinaremos el tipo de dato que representa (con la primera palabra `room`, `item`, ...) y se llamará a uno de los métodos `CreateItem, CreateRoom` con los parámetros adecuados obtenidos de la línea de texto leída. Para las descripciones, que van entrecomilladas y pueden contener blancos, usaremos un método

  ```
  private string ReadDescription(string linea)
  ```

  El método `ReadMap` lanzará la excepción correspondiente en el caso de que se produzca un error al leer el archivo. La excepción incluirá información sobre la línea del archivo en el que se ha producido el error y el motivo del error (la línea no empieza con una de las palabras esperadas, el número o tipo de argumentos no es el esperado, etc).

Serán necesarios también los métodos:

- `int FindItemByName(string itemName)`: devuelve el índice del objeto dado en el vector de `items`; -1 si no está.
- `int FindRoomByName(string roomName)`: devuelve el índice de la habitación en el vector de `rooms`; -1 si no está.
- `int GetItemWeight(int itemNumber)`: devuelve el peso del objeto indicado.
- `int GetItemHP(int itemNumber)`: devuelve los health points del objeto indicado.
- `string PrintItemInfo(int itemNumber)`: devuelve una cadena de texto con la información del objeto indicado.
- `string GetRoomInfo(int roomNumber)`: devuelve una cadena de texto con el nombre y la descripción de la habitación indicada.
- `string GetInfoItemsInRoom(int roomNumber)`: devuelve una cadena de texto con la información de los objetos que hay en el lugar indicado (o `I don't see anything notable here` si el lugar está vacío)
- `bool PickItemInRoom(int roomNumber, int itemNumber)`: coge el objeto indicado del lugar indicado. Si el objeto existe en la habitación y se ha podido coger, devuelve `true` (`false`, en caso contrario).
- `bool IsExit(int roomNumber)`: determina si el lugar indicado es una salida en el mapa.
- `int GetEntryRoom()`: devuelve el índice del lugar de entrada del mapa.
- `string GetMovesInfo(int roomNumber)`: devuelve una cadena de texto con la información de las salidas del lugar indicado.
- `public int Move(int roomNumber, Direction dir)`: devuelve el lugar al que se llega desde el lugar `roomNumber` avanzando en la dirección `dir` (-1 en caso de error).

A continuación implementaremos la clase `Player` para representar el estado del jugador:

```
namespace Adventure {
  class Player {
      string name;   // nombre del jugador
      int pos;       // lugar en el que esta
      int hp;        // health points
      int weight;    // peso de los objetos que tiene
      Lista inventory;  // lista de objetos que lleva

      const int MAX_HP = 10;         // maximo health points
      const int HP_PER_MOVEMENT = 2; // hp consumidos por movimiento
      const int MAX_WEIGHT = 20;     // maximo peso que puede llevar
```

Implementará los siguientes métodos:

- `public Player(string playerName, int entryRoom)`: constructora de la clase. Crea un jugador con nombre `playerName` y lo sitúa en la posición `entryRoom`, con inventario vacío, peso 0 y puntos de salud al máximo `MAX_HP`.
- `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 bool Move(Map m, Direction dir)`: mueve el jugador en la dirección `dir` a partir de la posición actual de acuerdo con el `Map m` (utiliza el método `Move` de la clase `Map`). Además, reduce sus puntos de salud. Si el movimiento se ha podido ejecutar devuelve `true`; `false`, en caso contrario.

- `public void PickItem(Map m, string itemName)`: recoge el objeto `itemName` del lugar actual y lo inserta en el inventario del jugador. Lanza una excepción en caso de que no se pueda ejecutar, indicando el motivo (el nombre del objeto no existe, el objeto no está en la habitación o el objeto es demasiado pesado para el jugador porque supera el peso máximo junto con el resto de objetos del inventario, etc).

- `public void EatItem(Map m, string itemName)`: busca el objeto `itemName` en el inventario y se lo come, si existe y es comestible (hp del objeto es mayor que 0), obteniendo los hp que proporciona dicho objeto. Además lo elimina del inventario y reduce el peso que lleva el jugador. Lanza una excepción en caso de que no se pueda ejecutar, indicando el motivo (el nombre del objeto no existe, el objeto no está en el inventario o el objeto no es comestible porque no da puntos de salud, etc.).

- `public string GetInventoryInfo(Map m)`: devuelve una cadena de texto con información sobre los objetos que lleva el jugador (o `My bag is empty`, si el inventario está vacío).

- `public string GetPlayerInfo()`: devuelve una cadena de texto con la información del jugador: nombre, hps y peso que lleva.

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

- `static bool HandleInput(string com, Player p, Map m, bool quit)`: procesa el comando representado en la cadena `com`. Devuelve `quit=true` si el jugador quiere abandonar la partida. El método devuelve *true* si el comando es válido y se ha podido ejecutar correctamente; *false*, en caso contrario.

- `void InfoPlace(Map m, int roomNumber)`: escribe en pantalla la información del lugar `roomNumber` y los posibles movimientos.

- `bool ArrivedAtExit(Map m, Player thePlayer)`: comprueba si el jugador se encuentra en un lugar que es salida en el mapa.

- `Main`: crea un mapa, lo lee del archivo dado, solicita el nombre del jugador 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 `HandleInput` hasta alcanzar un punto de salida, hasta que el jugador aborte el juego (comando `quit`) o el jugador muera. Finalmente, se presenta un mensaje de finalización distinto en función del motivo por el que se haya completado el juego.

Extensiones opcionales:

Una vez implementadas las clases descritas, puede extenderse la funcionalidad del programa:

- Implementar el comando `dropItem` que permita soltar un objeto en la posición actual en el mapa (que podrá volver a cogerse posteriormente).

- Extender la implementación para poder interrumpir una partida y recuperarla posteriormente. Debe guardarse el estado actual del juego en un formato adecuado (que debe diseñar el alumno), para poder restaurarlo posteriormente.
- Implementar un método que permita leer de archivo una lista de comandos para el jugador (un comando por línea) y haga que jugador ejecute dichas instrucciones en secuencia.