Fundamentos de la programación 2

Práctica 2. Adventure game

Indicaciones generales:

- La línea 1 del programa (y siguientes) deben contener los nombres de los alumnos de la forma:

  `// Nombre Apellido1 Apellido2`

- **Lee atentamente el enunciado** e implementa el programa tal como se pide, con las clases, métodos, parámetros y requisitos que se especifican. No puede modificarse la representación propuesta, ni alterar los parámetros de los métodos especificados (a excepción de la forma de paso (`out`, `ref`, ...), que debe determinar el alumno). Pueden implementarse todos los métodos adicionales que se consideren oportunos, especificando claramente su cometido, parámetros, etc.

- El programa, además de correcto, debe estar bien estructurado y comentado. Se valorarán la claridad, la concisión y la eficiencia.

- La entrega se realizará a través del campus virtual, subiendo un único archivo .zip, con los archivos fuente `Program.cs, Map.cs, Room.cs, List.cs` (y otras fuentes, si fuese necesario).

- El **plazo límite para la entrega** es el miércoles 19 de abril.

Las aventuras conversacionales constituyen un género de juegos (también conocido como *Interactive Fiction*[1]) 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 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 que 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, hablar con otros personajes, etc.

En esta práctica implementaremos un *motor de juego* para una aventura conversacional basada en una propuesta de Eric Roberts, de la universidad de Stanford[2]. El jugador puede moverse por distintas habitaciones (*rooms*) recogiendo objetos (*items*) mediante comandos sencillos. Un ejemplo:

```
Outside building
You are standing at the end of a road before a small brick building.  A small stream flows out of th
e building and down a gully to the south.  A road runs up a small hill to the west.

> in
Inside building
You are inside a building, a well house for a large spring. The exit door is to the south.  There is
 another room to the north, but the door is barred by a shimmering curtain.

> take keys
Item KEYS taken

> north
Curtain1
A closed curtain. Only passable with the NUGGET.

Missing Treasures
You can pass through this curtain only if you're carrying all the treasures.  You don't yet have all
 six.

Inside building
You are inside a building, a well house for a large spring. The exit door is to the south.  There is
 another room to the north, but the door is barred by a shimmering curtain.

> inventory
Items:
  - KEYS: a set of keys
```

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

[2] `https://cs.stanford.edu/people/eroberts/courses/cs106a/handouts/57-assignment-6.pdf`

Empezamos en la habitación *Outside building*, para la que se muestra una descripción. El jugador teclea `in` en el prompt y accede a *Inside building*. Ahí recoge el ítem *KEYS* con el comando *take keys*. Después va al norte y accede a *Curtain1*. Esta habitación tiene definido un desplazamiento forzado a *Missing Treasures*, que a su vez tiene otro movimiento forzado a *Inside building*, que es donde finalmente queda el jugador (el archivo de definición de habitaciones define estos movimientos forzados, como veremos). En la última interacción con el comando *inventory* obtiene el conjunto de ítems que ha recogido hasta el momento.

1. Archivos de datos

El mapa de juego, viene dado en dos archivos, *CrowtherItems.txt* y *CrowtherRooms.txt*, que leera el programa. El primero contiene información sobre los ítems de juego en el siguiente formato:

```
KEYS
a set of keys
3

LAMP
a brightly shining brass lamp
8
...
```

Cada ítem tiene un nombre (`KEYS`), una descripción (`a set of keys`) y la habitación donde está inicialmente el ítem (`3`). El archivo *CrowtherRooms.txt* tiene la información de las habitaciones, cada una con el siguiente formato:

```
1
Outside building
You are standing at the end of a road before a small brick building.  A small stream
flows out of the building and down a gully to the south.  A road runs up a small hill
to the west.
-----
WEST       2
UP         2
NORTH      3
IN         3
SOUTH      4
DOWN       4
```

La primera línea es el número de la habitación (`1`, en este caso), la siguiente el nombre (`Outside building`), la siguiente una descripción (`You are standing...`). Después hay una línea de separación `-----` y a continuación una serie de líneas con las posibles rutas desde esa habitación: en la dirección `WEST` está la habitación `2`, igual que en la dirección `UP` (en esta habitación `WEST` y `UP` son direcciones sinónimas); en la dirección `NORTH` (o `IN`) está la habitación 3, etc.

Algunas habitaciones contienen *direcciones condicionales* que exigen que el jugador tenga un ítem dado en su inventario. Por ejemplo, en la habitación 9 tenemos:

```
9
Cobble crawl
You are crawling over cobbles in a low east/west passage. There is a dim light to the
east.
-----
EAST      8
WEST      12/LAMP
WEST      10
```

En este caso, la dirección `WEST` conecta con la habitación 12 si el jugador tiene el ítem `LAMP` y con la habitación 10 en otro caso. **Es relevante el orden en que aparecen las direcciones**.

Por otro lado, hay habitaciones con *rutas forzadas* como las siguientes:

```
70
Curtain1
A closed curtain. Only passable with the NUGGET.
-----
FORCED    71/NUGGET
FORCED    76

76
Missing Treasures
You can pass through this curtain only if you're carrying all the treasures.
You don't yet have all six.
-----
FORCED     3
```

En este caso, si el jugador accede a la habitación 70, automáticamente se verá desplazado a otra habitación: si tiene el ítem `NUGGET` irá a la habitación 71 y si no, a la 76. A su vez, desde la habitación 76 hay otro movimiento forzado a la 3.

2. Lectura de datos. Versión inicial

Vamos a comenzar el desarrollo *parseando* los archivos de datos de la sección anterior, por ahora sin almacenarlos en memoria. Implementaremos los siguientes métodos:

- `void ReadInventory(string file)`: lee los ítems del archivo `file` (invocaremos con *CrowtherItems.txt*) y escribe la información en pantalla de la forma:

  ```
  Item name: KEYS   Descr: a set of keys   InitRoom: 3
  Item name: LAMP   Descr: a brightly shining brass lamp   InitRoom: 8
  ...
  ```

- `void ReadRooms(string file)`: lee las habitaciones del archivo `file` (que será *CrowtherRooms.txt*). Para ello abre un stream de lectura `f` y a continuación, para cada habitación, lee el número e invoca al siguiente método auxiliar con ese stream `f`:

- `void ReadRoom(StreamReader f, int n)`: lee la información de la habitación `n` del stream `f` y escribe su información en pantalla. Por ejemplo, para la habitación 9:

  ```
  Room: 9  Name: Cobble crawl  Descr: You are crawling ...
  Route from room 9 to room 8, direction EAST. CondItem:
  Route from room 9 to room 12, direction WEST. CondItem: LAMP
  Route from room 9 to room 10, direction WEST. CondItem:
  ```

  Tal como se aprecia en el archivo, la definición de una habitación termina con una línea en blanco o con el fin de archivo. Serán útiles los métodos `s.Trim()` (elimina blancos del principio y el final del string `s`) y `s.Split(" ",StringSplitOptions.RemoveEmptyEntries)` (trocea el string `s` utilizando blanco como separador).

Más adelante extenderemos estos métodos para que almacenen la información en memoria de manera estructurada en vez de escribirla en pantalla.

3. Representación de datos en programa

Para los datos referentes al mapa de juego vamos a utilizar dos clases: `Room` para las habitaciones y `Map` para el mapa (que utilizará la clase `Room`). La información del mapa se leerá de los archivos (*CrowtherItems.txt* y *CrowtherRooms.txt*). Esta información no cambiará durante el juego, a excepción de la ubicación de los ítems, que el jugador puede coger (añadir a su inventario) y soltar en distintas habitaciones. Así pues, para las habitaciones, las rutas y los ítems del juego vamos a utilizar arrays. Para el conjunto de ítems de las habitaciones y del inventario del jugador, que sí cambiarán durante el juego, utilizaremos listas de enteros (clase `List` explicada en clase), que se interpretan como índices al array de ítems.

Para las habitaciones definimos la clase:

```
class Room {
  struct Route { // tipo para las rutas
      public string direction;
      public int destRoom,        // habitación destino de la ruta
                 conditionalItem; // índice del ítem condicional (al array de ítems de Map)
  }                               // -1 si no hay ítem condicional

  string name, description; // nombre y descripción de la habitación leídos de CrowtherRooms
  Route [] routes; // array de rutas de la habitación
  int nRoutes;     // número de rutas = índice a la primera ruta libre
  List items;      // lista de índices de ítems (al array de ítems de Map)
  ...
```

Los campos `name` y `description` corresponden a los datos del mismo nombre que se leerán del archivo *CrowtherRooms.txt*. El campo `routes` es un array de rutas (tipo `Route`), cada una con su dirección, el número de la habitación destino y el índice del ítem condicional si se requiere en esa dirección (-1 si no se requiere). El número de destino corresponde al índice en el array de habitaciones contenido en `Map`, definida a continuación.

La clase `Map` será de la forma:

```
class Map {
  struct Item{  // información de cada ítem
      public string name, description; // como aparecen en el archivo CrowtherItems
      public int initialRoom; // índice de la habitación donde está al principio del juego
  }
  Room [] rooms; // array de habitaciones indexadas con la numeración de CrowtherRooms
  int nRooms;    // número de habitaciones = índice a la primera posición libre en rooms
  Item [] items; // array de items en el juego indexados por orden de aparición en el archivo
  int nItems;    // número de ítems = índice  la primera posición libre en items
  int maxRoutes; // número máximo de rutas por habitación
  ...
```

Para cada ítem, además del nombre y la descripción, guardaremos la habitación donde está ubicado inicialmente. Tendremos un array `rooms` de objetos de tipo `Room` para las habitaciones y un array para los ítems del juego. Gráficamente, el mapa en memoria quedará según se muestra en la Figura 1.

Nótese que la numeración de las habitaciones en el archivo *CrowtherRooms.txt* comienza en 1, por lo que la componente 0 del array `rooms` no se utiliza (la habitación virtual 0 representará la salida del mapa). Los ítems no están numerados en el archivo *CrowtherItems.txt* por lo que se añaden al array *items* en orden de aparición, de 0 en adelante (esté índice se utilizará en las listas de ítems).

4. Métodos de `Room` y `Map`, versión inicial

Para la clase `Room` implementaremos los siguientes métodos públicos:

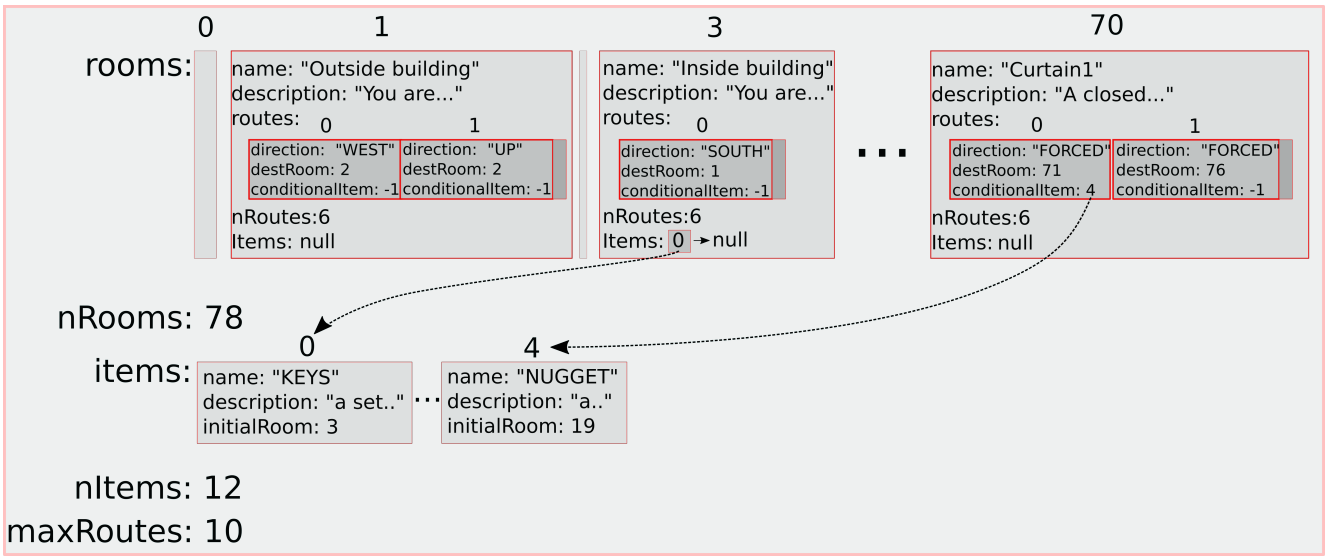

Figura 1: Representación del mapa en memoria

- constructora `Room(string nam, string des, int maxRts)`: inicializa la habitación con el nombre y la descripción dados; crea el array de rutas de tamaño `maxRts` con 0 rutas y una lista vacía de ítems.
- `void AddRoute(string dir, int desR, int condIt)`: añade una nueva ruta a la habitación con la dirección, destino e ítem condicional dados.
- `void AddItem(int it)`: añade el (índice del) ítem `it` a la lista de ítems de la habitación.
- `string GetInfo()`: devuelve una cadena de texto con el nombre y la descripción de la habitación.
- `int [] GetArrayItems()`: devuelve un array con los índices de los ítems de la habitación. De este modo se podrán ver los ítems que hay en una habitación, pero no modificarlos. Si se devolviese la lista podría cambiarse su contenido desde fuera, lo cual no es deseable.

  Nota: es necesario extender la clase `List` con un método público `ToArray` que devuelva un array con los números de la lista.

Para la clase `Map` implementaremos los métodos (todos públicos salvo `GetItemIndex`):

- constructora `Map(int maxRooms=100, int maxRts=10, int maxItems=20)`: crea los arrays `rooms` e `items` de tamaños `maxRooms` y `maxItems`, con 0 habitaciones y 0 ítems. Además inicializa el atributo `maxRoutes=maxRts`.
- `void AddItemMap(string name, string description, int iniRoom)`: añade un nuevo ítem al array de ítems con el nombre, la descripción y la habitación inicial dados. **Nota**: este método añade el ítem al mapa (al array de ítems), pero no lo coloca en la habitación correspondiente. En el momento de añadir un ítem al mapa, la habitación inicial del mismo puede no estar aún creada en el mapa. Como veremos, cuando estén leídas todas las habitaciones y los ítems, tendremos un método `SetItemsRooms` que colocará los ítems en las habitaciones de partida.
- `private int GetItemIndex(string name)`: busca el ítem `name` en el array de ítems y devuelve su posición en dicho array; -1 si no existe tal ítem.
- `void AddRoom(int nRoom, string name, string description)`: añade la habitación `nRoom` al mapa con el nombre y la descripción dados.

- `void AddRouteRoom(int nRoom, string dir, int destRoom, string condItem)`: añade a la habitación `nRoom` una nueva ruta con dirección `dir` y habitación destino `destRoom`. El nombre del ítem condicional viene dado como string (cadena vacía si no hay tal ítem); hay que obtener su índice con `GetItemIndex` para invocar al método `AddRoute` de la clase `Room`.
- `void AddItemRoom(int nRoom, int itemId)`: añade el ítem `itemId` a la habitación `nRoom`.
- `string GetInfoRoom(int nRoom)`: devuelve una cadena de texto con el nombre y la descripción de la habitación `nRoom`.
- `string GetItemsRoom(int nRoom)`: devuelve un string con la información de los ítems de la habitación `nRoom`.

5. Lectura y almacenamiento de datos

Una vez que tenemos la representación de datos, modificaremos los métodos de lectura de la sección 2 para generar el mapa de juego:

- `void ReadInventory(string file, Map map)`: lee los ítems de `file` y los almacena en `map` con `AddItemMap`.
- `void ReadRooms(string file, Map map)`: lee las habitaciones de `file` y las almacena en el mapa, utilizando el método siguiente:
- `void ReadRoom(StreamReader f, int nRoom, Map map)`: lee la habitación `nRoom` del flujo `f` almacena la información en `map` con los métodos `AddRoom` y `AddRouteRoom`.

Ahora extenderemos la clase `Map` con dos nuevos métodos:

- `void SetItemsRooms()`: recorre el array de `items` del mapa, añadiendo cada uno a su habitación de inicio. Una vez leídos los archivos de ítems y de habitaciones, este método se utilizará para terminar de inicializar el mapa colocando los ítems en las habitaciones correspondientes.
- `void WriteMap()`: escribe en pantalla toda la información del mapa. Para cada habitación escribe su nombre y descripción, las direcciones con su información correspondiente, así como los ítems que hay en cada habitación. Este método será util para depuración, para comprobar que el mapa se ha leído correctamente.

Para comprobar el funcionamiento, en el método `Main` haremos (por este orden): crear mapa, leer el inventario, leer las habitaciones, llamar a `SetItemsRooms` para colocar los ítems, llamar a `WriteMap` para ver el mapa almacenado.

6. Acciones del jugador

Para implementar las acciones de movimiento del jugador y la manipulación de ítems extenderemos la clase `Room` con los métodos (públicos):

- `int Move(string dir, List inventory)`: devuelve la habitación de destino en la dirección `dir`, si es posible el movimiento. Para ello busca la primera ruta en esa dirección que no requiera ítem condicional, o bien, requiera un ítem presente en la lista *inventory*. Si existe tal ruta devuelve la habitación de destino correspondiente; en otro caso devuelve -1.

- `bool ForcedMove()`: comprueba si la habitación tiene al menos una ruta y es `"FORCED"` (por definición, si una es forzada, todas deben serlo).
- `bool RemoveItem(int it)`: elimina el ítem de índice *it* de la lista de ítems de la habitación, si existe. En ese caso devuelve `true`, en otro caso `false`.

En la clase `Map` añadiremos los métodos (públicos):

- `bool TakeItemRoom(int nRoom, string itemName, Lista inventory)`: busca el ítem de nombre `itemName` en la habitación `nRoom`. Si está lo elimina de dicha habitación, lo añade a `inventory` y devuelve `true`; en otro caso devuelve `false`.
- `bool DropItemRoom(int nRoom, string itemName, List inventory)`: busca el ítem de nombre `itemName` en `inventory`. Si está lo elimina de dicha lista, lo añade a la habitación `nRoom` y devuelve `true`; en otro caso devuelve `false`.
- `List Move(int nRoom, string dir, List inventory)`: intenta el movimiento desde la habitación `nRoom` en la dirección `dir` y devuelve una lista con las habitaciones visitadas al hacer ese movimiento (nótese que puede ser más de una debido a los movimientos forzados). Para ello intenta el primer movimiento; si es posible, mientras la habitación destino sea de movimiento forzado, realiza los siguientes movimientos y va guardando los sucesivos números de habitación en una lista, que devolverá al final.
- `string GetItemsInfo(List inventory)`: devuelve un string con el nombre y la descripción de los ítems de `inventory`.

Con estos métodos ya podemos completar la implementación del juego. El jugador comenzará en la habitación 1 con el inventario vacío y terminará cuando alcance la habitación 0 (que en realidad no existe en el mapa). En cada iteración del bucle principal mostrará el prompt >, leerá el `input` (string) e invocará al siguiente método:

- `void ProcessCommand(Map map, string input, int playerRoom, List inventory)`: en primer lugar trocea el input en palabras sueltas:

  ```
  string [] words = input.Trim().ToUpper().Split(" ",StringSplitOptions.RemoveEmptyEntries);
  ```

  A continuación implementa cada una de las opciones en función de `words`:

  - `HELP`: muestra la ayuda del juego
  - `INVENTORY`: muestra el inventario actual del jugador
  - `LOOK`: muestra la información de la habitación actual
  - `ITEMS`: muestra los ítems de la habitación actual
  - `TAKE <item>`: si el `item` está en habitación actual lo recoge y lo añade al inventario del jugador; mensaje de error en otro caso
  - `DROP <item>`: si el `ítem` está en el inventario del jugador, lo elimina del inventario y lo deja en la habitación actual; mensaje de error en otro caso
  - si el comando no se ajusta a ninguno de los anteriores, se interpreta como dirección de movimiento, que se gestionará con el método correspondiente de `Map`.

Por último, se incluirá el **control de errores mediante excepciones** en la lectura de archivos. También pueden utilizarse para gestionar posibles inconsistencias del mapa (por ejemplo, al situar un ítem o establecer una ruta en una habitación inexistente, añadir una habitación con número duplicado, etc).

7. Extensiones opcionales

Pueden implementarse la siguientes extensiones:

- Procesado de comandos de archivo: puede implementarse fácilmente un método que lea comandos de un archivo dado (uno por línea, por ejemplo) y los ejecute en secuencia.
- El método anterior puede ser útil para guardar/restaurar partida: durante el juego se guardarán en memoria los comandos que lanza el jugador, que podrán salvarse en un archivo al acabar el juego. El método anterior permitirá restaurar el estado de la partida.
- Autocompletado de comandos: en los prompts textuales es útil tener una opción de autocompletado con el tabulador: se teclean uno o varios caracteres y con el tabulador se buscan comandos con ese prefijo; si el comando es único (no hay ambigüedad) se completa, si hay varios se dan las opciones.
- Sinónimos: en la versión original de este juego se cuenta con un archivo de palabras sinónimas que pueden utilizarse indistintamente para identificar los ítems.