



Práctica 4:

Motor de Aventuras Conversacionales v1.5

Fecha de entrega: 10 de marzo de 2011, 23:55 horas

Esta práctica consiste en **ampliar la versión 1.0 del motor de aventuras conversacionales** desarrollado en la práctica anterior con algunas funcionalidades nuevas (como el uso de objetos para bloquear y desbloquear las conexiones entre localizaciones) y soporte completo de los juegos definidos tanto en formato de texto plano como en formato derivado de XML. A partir de esta práctica la responsabilidad de seguir corrigiendo errores y mejorando el diseño (haciendo un uso razonable de los patrones de diseño) recae enteramente sobre los alumnos, quienes contarán con unas especificaciones de alto nivel sobre los requisitos funcionales que debe cumplir en la entrega, algunos de ellos obligatorios y otros opcionales.

Todo lo relativo a los métodos y las herramientas de trabajo, así como al sistema de corrección y pruebas se mantiene similar a prácticas anteriores, solo que ahora **todas las pruebas automáticas y los juegos de ejemplo serán proporcionados por el alumno**. Durante la evaluación, el profesor valorará el tipo de pruebas realizadas y podrá modificar los ejemplos para comprobar la veracidad de los mismos y la robustez del funcionamiento del motor.

Se recuerda que el alumno debe crear un proyecto nuevo para cada práctica, conservando intacto el código y los ejemplos de todas las entregas anteriores.

1. Descripción de la parte obligatoria

Además de toda la funcionalidad que hemos ido añadiendo hasta ahora, el motor tendrá que ser extendido hasta cumplir los siguientes requisitos.

Objetos con peso

Todos los objetos del juego tendrán asociado, además de un valor, otro número entero que será su **peso**. El inventario del jugador tendrá una **capacidad límite**,

por defecto de 10 unidades, que representa el número máximo de kilogramos que el personaje es capaz de portar consigo en su aventura. Este límite se tendrá en cuenta al coger o soltar un objeto, pudiendo ser denegada la acción por el juego si su realización implicase rebasar dicho límite. En ese caso se mostraría el correspondiente mensaje informando al jugador de que su orden ha fracasado.

Obstáculos sofisticados para moverse entre localizaciones

Cada conexión entre una localización y otra es susceptible de tener asignado un **obstáculo**. Este obstáculo se representará como un elemento del juego que admite dos estados posibles: activado y desactivado, de modo que el juego impedirá que un jugador vaya de una localización a otra cuando en esa conexión exista un obstáculo y este se encuentre activado. Para explicar ese fracaso al intentar moverse, los obstáculos tendrán además un mensaje que se mostrará al jugador como explicación del evento ocurrido. Además el obstáculo podrá contener dos listas de elementos asociados, ambas totalmente opcionales e independientes. Una primera lista contendría los **identificadores de aquellos objetos vinculados** que el jugador –como se explica más adelante– podrá usar para activarlo o desactivarlo. Serían por así decirlo las “llaves” que con las que poder abrir dicha “puerta”. La segunda lista contendría los **identificadores de otros obstáculos vinculados** cuyo estado deben invertirse cada vez que el estado del obstáculo en cuestión varíe. La utilidad más obvia es la de simular el comportamiento de una puerta que separa dos habitaciones en el mundo real: cuando se desactiva un obstáculo existente entre A y B, se provoca el cambio de otro obstáculo análogo entre B y A, que también estaría activado y pasa ahora a estar desactivado; y viceversa. Sin embargo nótese que hay utilidades más sofisticadas, ya que se trata de una lista de N obstáculos vinculados que *no* tienen porque encontrarse en el mismo estado que el obstáculo afectado por el cambio original. Por ejemplo podemos encontrarnos con una estructura circular que debemos tratar de manera que no se entre en un bucle infinito de cambios.

Apertura y cierre de obstáculos

El jugador ahora dispone de cuatro órdenes fundamentales para cambiar el estado de los obstáculos, que son:

- OPEN <dirección>: Orden de acción que cambia el estado del obstáculo existente en la dirección <dirección> a **desactivado**. Esta orden puede fallar por varios motivos (que no haya localización en esa dirección, que no haya obstáculo, que el obstáculo ya esté desactivado o que el obstáculo tenga asociado algún objeto vinculado), lo que se indicará con el correspondiente mensaje.

- CLOSE *<dirección>*: Orden de acción que cambia el estado del obstáculo existente en la dirección *<dirección>* a **activado** y que funciona de manera análoga a OPEN.
- OPEN *<dirección>* WITH *<nombreObjeto>*: Orden de acción que cambia el estado del obstáculo existente en la dirección *<dirección>* a **desactivado** usando el objeto de nombre *<nombreObjeto>*. Esta orden puede fallar aún por más causas (por ejemplo porque el jugador no tenga en su poder un objeto con dicho nombre, o porque el obstáculo no tenga objetos vinculados o porque sí los tenga pero ese objeto no sea uno de ellos...) y también mostrar los correspondientes mensajes informativos al jugador.
- CLOSE *<dirección>* WITH *<nombreObjeto>*: Orden de acción que cambia el estado del obstáculo existente en la dirección *<dirección>* a **activado** y que funciona de manera análoga a OPEN WITH.

Definiciones de juego en formato XML

El motor aceptará las definiciones del juego en formato de texto plano y en formato de texto estructurado según un lenguaje derivado de XML cuya gramática especificaremos con la correspondiente DTD. El aspecto de los **juegos definidos en texto plano** se mantendrá en la línea de lo hecho hasta ahora, tratando de ofrecer una alternativa lo más cómoda posible para el autor de aventuras conversacionales. Y para definir el aspecto que tendrán los **juegos definidos en XML** se puede tomar este ejemplo como punto de partida:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE game SYSTEM "game.dtd">
<!-- Game01: An Example Game. Comments about the authors, etc. -->

<game title="The title" author="The authors">
  The textual description of the game...
  <location id="entrance" name="The name">
    The textual description of this location...
    <item id="ticket" name="A ticket" value="25" weight="0">
      The textual description of this item...
    </item>
    ...
    <connection dir="s" target="hall" />
    <connection dir="e" target="corridor">
      <obstacle id="doorA" active="true">
        The textual message of finding this obstacle...
        <item-ref id="keyA" />
        <item-ref id="keyB" />
        ...
        <obstacle-ref id="doorB" />
        <obstacle-ref id="doorZ" />
        ...
      </obstacle>
    </connection>
    ...
  </location>
  ...
</game>
```

2. Descripción de la parte opcional

También se propone que, de manera opcional, el alumno extienda el motor incorporando todas o algunas de las siguientes propuestas.

Configuración avanzada del motor

El motor aceptará un **fichero de texto como configuración** (estructurado según la clase *Properties* de Java) donde, además de poderse cambiar los textos en inglés que el motor muestra por defecto al jugador y configurar también el funcionamiento por defecto de las órdenes de este, será posible cambiar estos otros atributos:

- **flag.showItemsValue** y **flag.showItemsWeight**: Valores booleanos que indican, respectivamente, si se debe mostrar o no el valor y el peso de los objetos al ser descritos o examinados.
- **flag.showConnections**: Valor booleano que indica si el motor debe mostrar o no el listado de las direcciones posibles a donde moverse desde la localización del jugador.
- **flag.showConnectionsState**: Valor booleano que indica si el motor debe mostrar o no el estado de las conexiones con otras localizaciones. Las posibilidades son: **clear** (si está despejada de obstáculos), **open** o **closed** (según el estado del obstáculo).
- **limit.inventoryCapacity**: Valor entero que indica el peso máximo en objetos que puede portar el jugador en su inventario.

Sistema para deshacer y rehacer órdenes ejecutadas

Además de permitir la **ejecución consecutiva de varias órdenes UNDO**, el jugador tendrá la posibilidad de “rehacer” las últimas órdenes “deshechas” con la orden REDO. El motor también debería asegurarse de que el estado del juego vuelve realmente a la situación deseada, por lo que no es aconsejable tratar de ejecutar órdenes “inversas” a las que se pretenden deshacer (ya que puede que, por ejemplo, no haya obstáculos de A a B pero sí los haya de B a A). La mejor forma de asegurar esto es usar un patrón de diseño conocido como *Snapshot*, que consiste en hacer copias “inertes” de toda la información del juego en un momento dado, de manera que lo que en realidad hace UNDO es **volver a un estado anterior del juego**, machacando el estado actual; es decir, que en vez de guardar un historial de órdenes ejecutadas, guardaríamos un historial de estados de juego previos.

El juego deberá distinguir, además, entre situaciones que modifican el mundo (como la ejecución de TAKE o DROP, cuando tienen éxito) y situaciones que no lo modifican (como cualquier ejecución de LOOK), ya que sólo tiene sentido guardar el estado de juego previo en el historial si este ha sido modificado.

Información de ayuda para el jugador más completa

En vez de limitarse a escribir por pantalla la plantilla de cada una de las órdenes disponibles para el jugador en el juego, el motor debería mostrar **más información de utilidad** como:

- **Configuración activa en el juego:** Información sobre el estado de los diferentes atributos configurables del motor y cómo repercute eso en la jugabilidad.
- **Historial de órdenes ejecutadas:** Información estadística sobre el número de órdenes de cada tipo que se han ejecutado, y su porcentaje en función del total, más el número de órdenes de “deshacer” consecutivas que es posible realizar en este momento del juego.
- **Ayuda específica del juego:** Consejos y pistas, creadas por el propio autor del juego e incluidas en el fichero de definición del mismo, sobre cómo disfrutar el juego y superar los retos que presenta al jugador.

3. Pruebas del código

Los alumnos son ahora responsables de desarrollar todas las pruebas unitarias (clase a clase, método a método) e integrales (funcionalidad completa) del código específico de su proyecto, siguiendo el procedimiento de las prácticas anteriores. Sólo será obligatorio desarrollar un *conjunto mínimo de pruebas* que verifiquen las partes más vulnerables del proyecto, siendo opcional el hacer pruebas exhaustivas de toda la API (incluyendo el correcto lanzamiento de excepciones) y de los casos más triviales de posibles ejecuciones.

La cantidad y sobretodo la calidad de las pruebas serán factores a valorar de cara a la calificación final.

4. Juegos de ejemplo y ficheros asociados

Los alumnos deben desarrollar **al menos un juego** a modo de ejemplo, lo suficientemente complejo como para probar el funcionamiento de todas las características del motor que han sido implementadas desde el comienzo del curso. Para ello deberán incluir **al menos tres ficheros diferentes de entrada, con sus correspondientes ficheros de salida**, que sirvan para realizar tres ejecuciones completas del juego en modo automático.

La variedad y sobretodo la complejidad de los juegos de ejemplo y sus ficheros asociados serán factores a valorar de cara a la calificación final.

5. Instrucciones de entrega y corrección

La práctica deberá ser entregada usando los mecanismos habituales para entregas de prácticas a través del Campus Virtual. Esto se realizará no más tarde de la fecha y hora que figuran en la cabecera de este documento.

Sólo uno de los miembros del grupo realizará la entrega, subiendo al Campus Virtual un fichero comprimido en formato **ZIP** llamado **lpsX-gNN.zip** siendo **X** el número de práctica y **NN** el número de grupo expresado con dos dígitos.

El contenido del fichero será **una carpeta de compilación y pruebas automáticas** similar a la que hemos utilizado en la práctica anterior, y en la que ya estarán incluidos y convenientemente editados todos los ficheros necesarios para la corrección por parte del profesor, automatizando la ejecución de las pruebas y ejemplos que proporciona el alumno. Además de todos los ficheros comunes a esta infraestructura, la carpeta debe contener:

- Directorio **src** con el código fuente Java de la práctica.
- Directorio **test** con el código fuente Java de todas las pruebas.
- Un directorio **gameZZ** por cada juego de ejemplo desarrollado por los alumnos; siendo **ZZ** la numeración consecutiva que se les asigna (01, 02, 03, ...). Además en cada directorio se añadirán los posibles ficheros de configuración aplicables a ese juego y los ficheros de entrada y de salida (convenientemente numerados) pensados para automatizar varias ejecuciones del mismo.

La corrección se realizará en la sesión de laboratorio posterior a la fecha de la entrega y deberán asistir todos los integrantes del grupo para proceder a la defensa individual de su trabajo mediante una entrevista con el profesor.