Práctica 1: Buffy the Vampire Slayer

Fecha de entrega: 10 de Noviembre de 2020, 09:00

Objetivo: Iniciación a la orientación a objetos y a Java; uso de arrays y enumerados; manipulación de cadenas con la clase `String`; entrada y salida por consola.

Control de copias

Durante el curso se realizará control de copia de todas las prácticas comparando las entregas de todos los grupos de TP. Se considera copia la reproducción total o parcial de código de otros alumnos o cualquier código extraído de Internet o de cualquier otra fuente, salvo aquellas autorizadas explícitamente por el profesor.

En caso de detección de copia se informará al Comité de Actuación ante Copias que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres siguientes medidas:

- Calificación con cero en la convocatoria de la asignatura a la que corresponde la prueba.
- Calificación con cero en todas las convocatorias del curso actual.
- Apertura de expediente académico ante la Inspección de Servicios de la Universidad.

1. Introducción

Vamos a crear un juego de vampiros inspirado en los clásicos Tower Defense, en los que el jugador puede colocar una serie de personajes con diferentes atributos para defenderse de los ataques de unos vampiros.

"La defensa de torres (en inglés Tower Defense), también llamado videojuegos de defensa de torres o videojuegos de defensa, es un subgénero del videojuego de estrategia en el que el objetivo es defender los territorios o las posesiones de un jugador mediante la obstrucción de los atacantes enemigos, que generalmente se logra mediante la colocación de estructuras defensivas alrededor y en su trayectoria de ataque, a pesar del nombre del

Figura 1: Portada del juego

género los elementos defensivos no siempre son torres típicas, pudiendo ser en realidad una gran gama de elementos defensivos, como personajes, estructuras, edificios de distinto tipo, elementos minerales o vegetales, entre otros."

A nivel visual el juego se presenta como una cuadrícula en la que el jugador coloca cazavampiros o *slayers* con diferentes características; los vampiros aparecen de un lado del tablero y avanzan hacia el lado del jugador que tiene que resistir su ataque, destruyéndolos todos. En el caso de que uno de los vampiros supere la defensa y llegue hasta el final del tablero, el usuario pierde la partida. Durante el cuatrimestre vamos a ir desarrollando el juego progresivamente. Empezaremos con una versión muy reducida e iremos incrementando su complejidad añadiendo nuevos tipos de slayers y vampiros con diferentes habilidades.

En la primera práctica tenemos como objetivo implementar la mínima versión jugable con la consola.

2. Descripción de la práctica

En nuestra primera práctica vamos a considerar que el juego consta de un tablero de $X \times Y$ casillas (X columnas por Y filas). El número exacto de casillas dependerá del nivel del juego. En cada casilla solo puede haber un slayer o un vampiro. Los vampiros van a avanzar de derecha a izquierda y ganan la partida cuando llegan al final del tablero. Inicialmente el tablero comienza vacío, el juego finalizará, si al final de un turno, todos los vampiros son destruidos o si por el contrario uno de los vampiros consigue llegar al final del tablero.

En la primera práctica solo hay dos tipos de objetos de juego, se darán detalles más abajo.

- ***Slayer***. Lo añade el usuario en una posición del tablero determinada. Cuestan 50 monedas. No se mueven y en cada ciclo de juego disparan balas de plata[1] y quitan un punto de vida al primer vampiro que esté en su misma fila.

- ***Vampiro***. Aparecen aleatoriamente en la última columna del tablero. Avanzan una casilla cada dos turnos, siempre y cuando no tengan nada delante. En caso de tener delante un slayer, le restan un punto de vida en cada turno.

En cada ciclo del juego se realizan secuencialmente las siguientes acciones:

1. ***Draw***. Se pinta el tablero y se muestra la información del juego.
2. ***User action***. El usuario realiza su acción (añadir nuevo slayer, pasar turno,...)
3. ***Update***. Se mueven los objetos que están en el tablero.
4. ***Attack***. Se comprueban los ataques y se resta la vida.
5. ***Add vampires***. Se añaden vampiros al azar.
6. ***Remove dead objects***. Se eliminan los cadáveres del tablero.
7. ***Check end***. Se analiza si el juego ha terminado.

Inicialmente el jugador comienza con 50 monedas con la que podrá comprar Slayers. En cada turno el usuario recibirá aleatoriamente 10 monedas con un 50 % de posibilidades.

3. Objetos del juego

En esta sección describimos el tipo de objetos que aparecen en el juego y su comportamiento.

Slayer

- **Comportamiento**: Dispara balas de plata a los vampiros de su misma fila.
- **Coste**: 50 monedas.
- **Resistencia**: 3 puntos de daño.
- **Frecuencia**: 1 disparo por ciclo.
- **Daño**: Cada disparo proporciona 1 punto de daño.
- **Alcance**: Solo dispara recto y hacia adelante.
- Se representan con la letra S

[1] En el juego no visualizamos los disparos

Vampiro

- **Comportamiento**: Avanza si puede, muerden a los slayers que tiene justo delante.
- **Resistencia**: 5 puntos de daño.
- **Daño**: 1 punto de daño.
- **Velocidad**: 1 casilla cada 2 ciclos.
- Se representan con la letra V

4. Actions

A continuación describimos lo que ocurre en cada parte del bucle del juego. El orden de las acciones es el especificado más arriba, en esta sección damos detalles de cada una de las partes.

4.1. Draw

En cada ciclo se pintará el estado actual del tablero; así como el ciclo de juego en el que nos encontramos (inicialmente 0), el número de monedas acumulados y el número de vampiros que hay en el tablero y el número que quedan por aparecer. El tablero se pintará por el interfaz consola utilizando caracteres ASCII, como muestra el siguiente ejemplo.

```
Number of cycles: 20
Coins: 50
Remaining vampires: 1
Vampires on the board: 2

 ---------------------------------------------------------------
|       | S [3] |       |       |       |       |       |       |
 ---------------------------------------------------------------
|       |       |       |       | V [5] |       |       |       |
 ---------------------------------------------------------------
| S [3] |       |       |       |       | V [1] |       |       |
 ---------------------------------------------------------------
|       |       |       |       |       |       |       |       |
 ---------------------------------------------------------------


Command >
```

Al lado de cada objeto en el tablero (slayers o vampiros) aparece la vida que les queda (entre corchetes). También mostraremos el prompt del juego para pedir al usuario la siguiente acción

4.2. Update

Las actualizaciones que ocurren en cada ciclo son:

- El usuario recibe 10 monedas con probabilidad del 50 %
- Los vampiros avanzan cuando les toca.

4.3. Attack

- Los slayers disparan a los vampiros que estén en su misma fila.
- Los vampiros muerden a los slayers que estén a su alcance (inmediatamente delante en la misma fila).

4.4. Add Vampire

Cada nivel tiene una frecuencia diferente de aparición de vampiros. Los vampiros aparecen en la última columna, en una fila aleatoria. Si en esa casilla ya hay otro vampiro entonces no se coloca el nuevo vampiro[2].

La aparición de los vampiros tendrá un comportamiento pseudoaleatorio. Uno de los parámetros de entrada del programa será el nivel. Definiremos tres niveles EASY, HARD y INSANE (ver Tabla 1.1). Cada nivel determinará varias opciones de configuración del juego:

Nivel	**Número de Vampiros**	**Frecuencia**	**DIM X**	**DIM Y**
EASY	3	0.1	8	4
HARD	5	0.2	7	3
INSANE	10	0.3	5	6

Tabla 1.1: Configuración para cada nivel de dificultad

- El número total de vampiros que aparecen en la partida.
- La frecuencia de aparición de los vampiros. La frecuencia determina la probabilidad de que aparezca un vampiro en un ciclo dado. Así pues, si la frecuencia es 0.2, saldrá aproximadamente un vampiro cada 5 ciclos; aunque al tratarse de una probabilidad pueden salir más espaciados o menos según la aleatoriedad.

En la tabla de nivel también vienen definidas las medidas del tablero, así en el nivel más fácil el tablero será 8×4 y en el más difícil 5×6.

4.5. User command

Se le preguntará al usuario qué es lo que quiere hacer, a lo que podrá contestar una de las siguientes alternativas:

- add <x><y>: Este es un comando para añadir un nuevo slayer x, y. Obviamente el usuario solo podrá añadir un slayer por ciclo si tiene el número suficiente de monedas. No podrá añadir un slayer en una casilla ocupada por otro slayer o por un vampiro. Tampoco podrá añadir un slayer en la última columna porque esto bloquearía la aparición de nuevos vampiros.

[2]puede darse el caso que la casilla donde se iba a poner el vampiro ya esté ocupada, entonces habría dos opciones: buscar una casilla vacía para colocarlo o salta turno sin poner el vampiro. Hemos optado por la segunda opción

- reset: Este comando permite reiniciar la partida, llevando al juego a la configuración inicial.
- none: El usuario no realiza ninguna acción y el juego se actualiza. Si no se introduce nada (simplemente se pulsa el return el programa hace un update
- exit: Este comando permite salir de la aplicación, mostrando previamente el mensaje "Game Over".
- help: Este comando solicita a la aplicación que muestre la ayuda sobre cómo utilizar los comandos. Se mostrará una línea por cada comando. Cada línea tiene el nombre del comando seguida por ':'y una breve descripción de lo que hace el comando.

```
Command > help
Available commands:
[a]dd <x> <y>: add a slayer in position x, y
[h]elp: show this help
[r]eset: reset game
[e]xit: exit game
[n]one | []: update
```

Observaciones sobre los comandos:

- La aplicación debe permitir comandos escritos en minúscula y mayúscula o mezcla de ambos.
- La aplicación debe permitir el uso de la primera letra del comando en lugar del comando completo [A]dd, [N]one, [R]eset, [H]elp, [E]xit.
- Si el comando es vacío se identifica como none y se avanza al siguiente ciclo de juego.
- Si el comando está mal escrito, no existe o no se puede ejecutar, la aplicación mostrará un mensaje de error.
- En el caso de que el usuario ejecute un comando que no cambia el estado del juego o un comando erróneo, el tablero no se debe repintar.

El juego finalizará si al final de un turno todos los vampiros son destruidos o uno de los vampiros llega al final de la fila. Cuando el juego termine se debe mostrar quién ha sido el ganador: "Player wins" o "Vampires win". Siempre se actualizan los elementos del juego en el orden en el que fueron introducidos. Este orden de actualización no es el único posible, y el resultado final puede variar dependiendo del orden. Vamos usar todos el mismo para que los resultados sean consistentes.

Para controlar el comportamiento aleatorio del juego y poder repetir varias veces la misma ejecución utilizaremos una semilla, o como se conoce en inglés, seed. Este valor lo puede introducir o no el usuario proporciona un control del comportamiento del programa lo que nos permitirá repetir exactamente una misma ejecución. Si el usuario no introduce ninguna semilla entonces la semilla se genera automáticamente. Es muy importante que solo creemos un objeto Random en el ciclo de vida aplicación y que todos los objetos lo compartan para que los resultados sean reproducibles.

5. Implementación

La implementación propuesta para la primera práctica no es la mejor; ya que las vamos hacer sin utilizar **herencia** y **polimorfismo**, dos herramientas básicas de la programación orientada a objetos. Durante el curso veremos formas de mejorarla mediante el uso de las herramientas que nos brinda la programación orientada a objetos.

Para implentar la primera versión tendremos que copiar y pegar mucho código y esto casi siempre es una mala práctica de programación. La duplicación de código implica que va a ser poco mantenible y testeable. Hay un principio de programación muy conocido que se conoce como **DRY (Don't Repeat Yourself)** o **No te repitas** en castellano. Según este principio toda "pieza de información" nunca debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior, puede perjudicar la claridad y crear un espacio para posibles inconsistencias.

A lo largo de las siguientes prácticas veremos cómo refactorizar el código para evitar repeticiones.

5.1. Detalles de implementación

Para lanzar la aplicación se ejecutará la clase `org.ucm.tp1.BuffyVampireSlayer`, por lo que se aconseja que todas las clases desarrolladas en la práctica estén en paquetes que cuelguen de `org.ucm.tp1`. Recordad que según las convenciones de nomenclatura para Java, todos los paquetes deben ir en minúsculas, y las clases deben empezar siempre por mayúsculas.

Para implementar la práctica necesitarás, al menos, las siguientes clases:

- Slayer, Vampire: Estas clases encapsulan el comportamiento de los elementos del juego. Tienen atributos privados, como su posición x, y, su vida, etc... También tienen un atributo en el que almacenan una referencia al juego, eso es, una instancia de la clase Game (que será la única en el programa) que como veremos contiene la lógica del juego. De este modo, estas clases podrán usar los métodos de la clase Game para consultar si pueden hacer o no una determinada acción.

  - La gestión de *cuántos vampiros quedan por salir*, *cuántos vampiros están en el tablero* o si *los vampiros han llegado al final* se realizará desde la propia clase Vampire utilizando variables y métodos estáticos. El Game accederá a esos métodos para mostrar la información o para terminar el juego cuando sea necesario.

- Player: Esta clase sirve para llevar el control de los atributos propios del jugador, que de momento son sólo las monedas que tiene disponibles. Puedes también tener una referencia al generador de números aleatorios (instancia de Random) y así poder recurrir a él cuando toque calcular si se reciben o no monedas.

- VampireList, SlayerList: Contienen arrays de los respectivos elementos del juego, así como métodos auxiliares para su gestión. En esta práctica usaremos obligatoriamente arrays convencionales, en las siguientes prácticas las substituiremos por ArrayLists.

- GameObjectBoard: Contienen una VampireList, y una SlayerList con las referencias a los objetos del juego. Esta clase va a gestionar los accesos y llamadas a los objetos de sus listas, y Game delegará sobre ella las distintas funcionalidades.

- Game: Encapsula la lógica del juego. Tiene, entre otros, el método update que actualiza el estado de todos los elementos del juego. Contiene un GameObjectBoard al que delega la funcionalidad.

  Esta clase debe ser quien lleve el contador de ciclos de juego y también guardar una referencia a la instancia de Player.

- Level: Es un clase tipo Enum que contiene los valores correspondientes a cada nivel de juego.

- GamePrinter: Recibe el game y tiene un método toString que sirve para pintar el juego como veíamos anteriormente.

- Controller: Clase para controlar la ejecución del juego, preguntando al usuario qué quiere hacer y actualizando la partida de acuerdo a lo que éste indique. La clase Controller necesita al menos dos atributos privados:

  ```
  private Game game;
  private Scanner in;
  ```

  El objeto in sirve para leer de la consola las órdenes del usuario. La clase Controller implementa el método público public void run() que controla el bucle principal del juego. Concretamente, mientras la partida no esté finalizada, solicita órdenes al usuario y las ejecuta.

- BuffyVampireSlayer: Es la clase que contiene el método main de la aplicación. En este caso el método main lee los valores de los parámetros de la aplicación (nivel y semilla), crea una nueva partida (objeto de la clase Game), crea un controlador (objeto de la clase Controller) con dicha partida, e invoca al método run del controlador.

Observaciones a la implementación:

- Durante la ejecución de la aplicación solo se creará un objeto de la clase Controller. Lo mismo ocurre para las clases Game (que representa la partida en curso y solo puede haber una activa).

- Junto con la práctica os proporcionaremos unas plantillas con partes del código.

El resto de información concreta para implementar la práctica será explicada por el profesor durante las distintas clases de teoría y laboratorio. En esas clases se indicará qué aspectos de la implementación se consideran obligatorios para poder aceptar la práctica como correcta y qué aspectos se dejan a la voluntad de los alumnos.

6. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica.

El fichero debe tener al menos el siguiente contenido [3]:

- Directorio `src` con el código de todas las clases de la práctica.

- Fichero `alumnos.txt` donde se indicará el nombre de los componentes del grupo.

Recuerda que no se deben incluir los .class.

[3] Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse

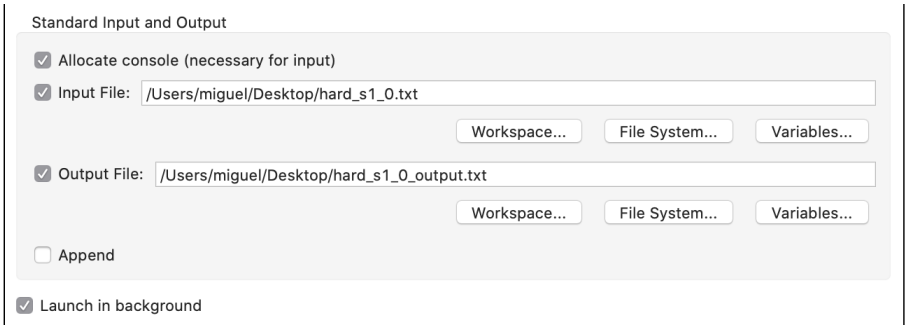

Figura 2: Redirección entrada y salida

7. Testing

Junto con la instrucciones de la práctica tendrás una carpeta con trazas del programa, encontrarás dos ficheros con la siguiente nomenclatura:

- *easy_s5_0.txt*: es la entrada del caso de prueba 0 can nivel *easy* y semilla 5.
- *easy_s5_0_output.txt*: es la salida esperada para la entrada anterior

En Eclipse, para usar un fichero de entrada y volcar la salida en un fichero de salida debes configurar la redirección en la pestaña *Common* de la ventana *Run Configurations* como te muestra la imagen.

Para nuestros tests hemos usado los siguientes métodos de la clase Random:

- nextDouble() para saber si hay que añadir un vampiro o no.
- nextInt() para saber en qué posición inicial sale el vampiro.
- nextFloat() para saber si en un ciclo el player recibe monedas o no.

Hay multitud de programas gratuitos para comparar visualmente ficheros, en la imagen se muestra *Beyond compare* (https://www.scootersoftware.com/) que se encuentra disponible para todas las plataformas y es muy útil para asegurarte de que tu salida corresponde con al deseada.[4] Ten mucho cuidado con el orden las instrucciones porque puede cambiar la salida considerablemente. Por supuesto, nuestra salida puede tener algún error, así que si detectas alguna inconsistencia por favor comunicanoslo para que lo corrijamos.

Durante la corrección de prácticas os daremos nuevos ficheros de prueba para asegurarnos que vuestras prácticas se generalizan correctamente, así que asegúrate de probar no solo los casos que te damos sino todas otras posibles ejecuciones que no estén presentes en ellos.

[4]Otro software que puedes usar es DiffMerge (https://sourcegear.com/diffmerge/)

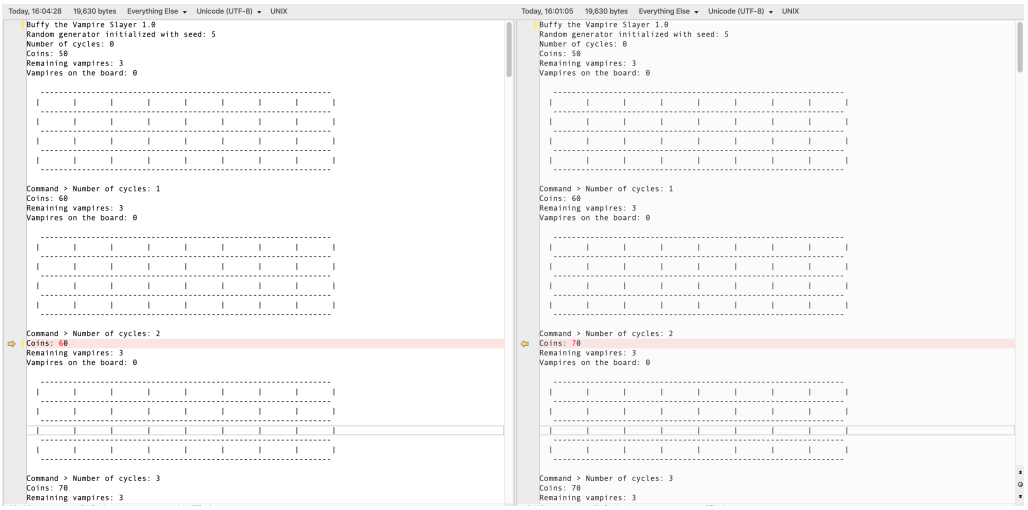

Figura 3: Compara salida obtenida con salida esperada