Entrega 4: Partida

El objetivo de esta última entrega es terminar de definir la jerarquía de las entidades que participan en el juego, adaptar la clase `Escenario` para que aplique los turnos del juego e implementar una clase que haga las funciones de controlador del juego y mediador con la ***interfaz gráfica de usuario*** (GUI): la clase **`Partida`**. Además, se deberá prestar especial atención a la ***robustez*** del proyecto, manejando las excepciones en los casos que sean necesarias.

1. Nuevas entidades

Hasta la entrega anterior los enemigos del juego parecen inmortales, ya que el jugador se encuentra desarmado ante ellos. Con el objetivo de que pueda eliminarlos y así poder avanzar por el escenario se introducirán **Armas** en el juego, que sólo las podrá coger el jugador. Todas las armas tienen un comportamiento en común: se pueden **disparar**. Disparar un arma significa lanzar un tipo de *proyectil* que está asociado con el arma. El disparo deberá realizarse en un sentido y el proyectil debe partir de la posición de quien dispara el arma, pero sin que el proyectil lo toque. Luego, todos los proyectiles tienen en común que se mueven en un sentido. Dispondremos de los siguientes tipos de armas:

- ***Cerbatana*:** dispara ***dardos*** sedantes. Cuando un dardo alcanza a un enemigo, además de restarle una vida, éste permanecerá inmovilizado un número de turnos hasta que se le pase el efecto del sedante. Durante este tiempo no pueden atacar al jugador. Cuando el dardo choca con cualquier entidad o sale del escenario desaparece.
- ***Tirachinas***: dispara ***piedras***. El alcance del disparo (desplazamiento total) dependerá de la fuerza con la que se dispare el tirachinas, y esta fuerza es proporcional a las vidas que tenga el jugador, esto es, cuantas más vidas tenga, con más fuerza disparará el tirachinas y más lejos llegará la piedra. Al final del desplazamiento la piedra queda depositada en el escenario como un obstáculo más. Si durante su trayectoria la piedra choca con alguna entidad termina su desplazamiento. Si choca con una entidad viva le restará una vida.
- ***Chicles***: dispara ***pompas***. El movimiento de una pompa es en el sentido del movimiento del jugador hasta que atrapa a un enemigo, momento en el cual comienza a ascender (y el enemigo con ella). Las pompas se pueden explotar de manera natural, al chocar con un obstáculo o al salirse del escenario, o pueden ser explotadas tanto por el jugador como por los enemigos. Si la pompa estaba vacía en el momento de la explosión, simplemente desaparece. Si la pompa tenía atrapado un enemigo y se explota por la acción de otro enemigo, el enemigo atrapado es liberado y recupera su movilidad natural. Por el contrario, si la pompa se explota de manera natural o por la acción del jugador, el enemigo atrapado muere (desaparece del escenario) y aparece en su posición un ***espíritu***. Un espíritu es una entidad inerte que se mueve de manera errática. Cuando un jugador ***atrapa*** a un espíritu éste desaparece y el jugador gana tantas vidas como vidas le quedasen al enemigo que ha muerto.

En el caso del tirachinas y la cerbatana se puede considerar que el número de proyectiles es ilimitado. Sin embargo, los chicles vienen en un paquete y se podrán hacer tantas pompas como chicles haya en dicho paquete.

Cuando un jugador coge un arma, ésta desaparece del escenario y pasa a formar parte del jugador. El jugador únicamente puede tener un arma en un momento dado, por lo que si el jugador se encuentra armado cuando coja otra arma debe dejar en el escenario la que llevaba hasta ese momento. Para que el usuario pueda disparar debemos incluir el método **`accionDispara`**. Por

tanto, el usuario podrá solicitar cuatro acciones al jugador: `mueveDerecha`, `mueveIzquierda`, `mueveSalta` y `accionDispara`. En su fase de actuación, el jugador comprueba si se ha solicitado realizar un disparo. Si es así y además dispone de arma, la *dispara*.

Con la introducción de las armas en el juego los enemigos pueden morir, es decir, desaparecen del escenario. Esto ocurre cuando su nivel de vida se agota. Este comportamiento es común a todas las entidades vivas.

Como consecuencia de la aparición de las pompas, debemos modificar el comportamiento de los enemigos para que acudan en ayuda de sus "compañeros" atrapados. Es decir, los enemigos deben estar observando el escenario para averiguar cuando una burbuja ha atrapado a otro enemigo, en ese caso, en lugar de quedarse quietos cuando su algoritmo de movimiento así lo establezca, deberán moverse para liberar al primer enemigo atrapado, o lo que es lo mismo, explotar la primera pompa que haya atrapado a un enemigo. Por su parte, el escenario es el objeto observado que tendrá que informar a sus observadores cuando cambia de estado.

2. El Escenario y los turnos

El propósito del juego es ofrecer la ficción de que las entidades están en continua actividad. Con este objetivo adaptamos la clase `Escenario` para que haga las funciones de planificador del juego y cuando le avise un temporizador recorra todas las entidades aplicando sus turnos. Para ello se hará uso del `Temporizador` visto en clase. El periodo de notificación es el factor que condiciona la velocidad del juego. Cuanto menor sea este periodo, más rápido irá el juego. Para poder poner en marcha el hilo que despierte al escenario para la asignación de turnos y poder pararlo (parando de este modo todas las entidades del juego) se deben implementar dos métodos nuevos en la clase `Escenario`: el método **`iniciar`** y el método **`parar`**.

3. Partida

La clase `Partida` representa la fachada del juego desde la interfaz de usuario. Es responsable de cargar los escenarios del juego y controlar los cambios de escenario del jugador. Además ofrece funcionalidad para actuar sobre los jugadores. Asimismo se encarga de controlar el tiempo empleado por el jugador en matar a todos los enemigos de un escenario.

Se supone que los escenarios residen en una carpeta de la aplicación con nombre "escenarios" y deben nombrarse utilizando números consecutivos. Esta secuencia de números establecerá el orden en el que se van a ir recorriendo, y en consecuencia el orden en el que se deben ir cargando los escenarios. Por tanto, la inicialización de un objeto partida consiste en cargar el escenario `0.esc` que ha de contener ***un jugador*** (para simplificar la GUI supondremos la existencia de un único jugador). Cuando el jugador mata a todos los enemigos de este escenario pasa al siguiente, cargando entonces el escenario `1.esc`. Y así sucesivamente hasta que al cargar el siguiente escenario, éste no se encuentre en el directorio "escenarios". Esto significará que ya hemos recorrido todos los escenarios y que, por tanto, hemos ***terminado la partida con éxito***.

La clase ofrece el método **`comenzar`** para arrancar la partida. En este método arranca un temporizador para controlar el *tiempo transcurrido en el escenario*, e inicia el primer escenario. Los objetos partida también ofrecen el método **`parar`** para detener la partida en curso, es decir, detener el escenario actual y el hilo temporizador, y otro **`reanudar`** para continuar con la ejecución.

Todos los escenarios tienen el mismo tiempo límite, de manera que si transcurre dicho tiempo sin que el jugador haya sido capaz de matar a todos los enemigos, la partida termina sin éxito. Por tanto, la partida puede ***terminar sin éxito*** bien por muerte del jugador o porque se haya agotado el tiempo establecido.

Asimismo queremos que esta clase ofrezca la funcionalidad necesaria para **`guardar`** y **`recuperar`** partidas en un fichero. Nótese que esta funcionalidad se podría implementar de diferentes formas. El propósito es utilizar **serialización**, lo cual simplificará notablemente el proceso. Téngase en cuenta además que la serialización funciona por alcance, de modo que si alguno de los objetos no es serializable no funcionará correctamente. También es importante reseñar que es posible que el estado completo de todos los objetos no sea almacenado. Esto último puede ocurrir cuando hacemos uso de clases que no hemos implementado (por ejemplo, la clase `Observable`). Finalmente, hay que recordar que los enumerados debemos adaptarlos para que funcionen con la serialización (ver apartado 5.2).

Interfaz de usuario

La última fase del desarrollo es ofrecer una interfaz de usuario al juego. Se ofrece una librería de clases gráficas que realizan la visualización el juego. Junto a la librería se incluye la documentación *javadoc* para su uso. Es el propósito de la práctica aprender a usar la documentación del código para averiguar cómo utilizar la librería (ver apartado 5.3).

4. Control de Errores

Hasta esta entrega hemos omitido el tratamiento de errores en el juego. Existen numerosas situaciones en las que el programa puede finalizar si existe un error y otras en las que hasta ahora es difícil tratar las situaciones de error.

De acuerdo con la técnica del Diseño por Contrato, los errores se producirán como consecuencia de la ruptura del contrato entre la rutina cliente y la rutina a la que invoca (rutina servidora), esto es, por fallo en la precondición, postcondición o en el invariante. Normalmente, un fallo en la precondición se debe a un error del cliente el cual invoca de manera incorrecta al método: un argumento no válido, con un valor nulo,… En este caso, el error se notificará lanzando una ***excepción no comprobada o Runtime*** (para las situaciones anterior se pueden utilizar las excepciones del lenguaje *IllegalArgumentException* y *NullPointerException* respectivamente). Por otro lado, un fallo en la postcondición se debe a que la rutina servidora, habiéndole pasado argumentos válidos (cumplen la precondición establecida) no puede terminar cumpliendo la postcondición. En ese caso, se debe informar a la rutina cliente que no se ha podido terminar la ejecución con éxito, bien devolviendo un valor especial, o bien lanzando una ***excepción, en este caso comprobada***. El optar por una u otra dependerá de cómo de importante es que el cliente no se olvide de manejar la situación de error y también de que en ocasiones no existe la posibilidad de devolver un valor especial, como es el caso de los constructores.

Respecto a los constructores, dos ejemplos claros de situaciones de error los encontramos en el Punto y el Rectángulo. El invariante de la clase Punto dice que las coordenadas no pueden ser negativas. En el caso del Rectángulo, existe error si los puntos no representan lo que son, es decir, las esquinas correspondientes, o si los puntos están alineados o son el mismo.

A continuación se relacionan algunas situaciones de error comunes que son notificadas con excepciones no comprobadas y que conviene tener en cuenta en el videojuego:

- Conversión de cadenas a enteros. Si la conversión no puede realizarse, el método no devuelve un valor especial, sino que lanza la excepción *NumberFormatException*.
- El método *nextToken* de un *StringTokenizer* lanza un error si no quedan elementos. Lo mismo sucede con el método *next* de los iteradores. La excepción que se produce es *NoSuchElementException*. Esta excepción se puede evitar comprobando si quedan elementos antes de aplicar las operaciones *next*.
- El método *substring* de las cadenas lanza la excepción *IndexOutOfBoundsException* si se invoca con índices fuera de rango. Esto puede suceder cuando se toma como valor de índice el resultado de una operación *indexOf* previa, que puede devolver un valor -1.

Otra de las excepciones Runtime predefinidas en el lenguaje Java es *IllegalStateException.* Esta excepción se utiliza para notificar que se está invocando un método en un momento que no es el adecuado, esto es, la aplicación no se encuentra en el estado adecuado para atender a la operación que se ha demandado. Por ejemplo, en el caso de la clase `Partida`, no tiene sentido invocar al método `parar` si la partida no ha comenzado.

El método `cargarEscenario` del Escenario es susceptible a errores de configuración del fichero. Este tipo de errores no dependen del programador, sino de los datos que proporciona el usuario externamente. Hasta ahora hemos tratado estos errores devolviendo un escenario nulo. Esta solución es válida, pero poco informativa. Es decir, un fichero de configuración puede ser muy extenso y convendría saber el motivo por el que no se ha podido crear: error de E/S, dimensiones incorrectas, no hay dimensiones, etc. Por tanto, crearemos la excepción comprobada **`ConfiguracionIncorrecta`** que notifique con detalle el error producido. Por otro lado, si las entidades no están correctamente definidas, hasta el momento se han descartado. Esto puede dar lugar a que el usuario obvie errores de configuración. Por este motivo, parece sensato también tratar estas situaciones como errores de configuración y notificarlos con la misma excepción.

Nótese que aunque no es obligatorio capturar las excepciones Runtime, en ocasiones puede ser necesario hacerlo para transformar la excepción no comprobada en otra que sea comprobada. Por ejemplo, capturar la excepción *NumberFormatException* para indicar que ha habido un error de configuración en el escenario.

En esta práctica hay que controlar las situaciones de error, prestando especial atención a la violación de las precondiciones de los métodos. En tal caso es conveniente utilizar las excepciones *Runtime* del lenguaje. En particular se deben definir en detalle las precondiciones de los métodos de las clases `Escenario` y `Partida`. Se valorará el control de las situaciones de error en el resto de clases del proyecto.

Además de tener en cuenta las precondiciones en el código (modificando los métodos para que chequeen las precondiciones y lancen la excepción *runtime* apropiada), debemos documentarlas en *javadoc* para lo que debemos definir una *plantilla de documentación* en JBuilder (ver apartado 5.4). Por ejemplo, el método `addEntidad` de `Escenario` quedaría como sigue:

```
/**
 * Añade una entidad al escenario
 * @param Entidad entidad
 *
 * <br><br><b>Precondiciones: </b><br>
 * <ul>
 *   <li> La entidad no puede ser nula
 * </ul>
*/
public boolean addEntidad (Entidad entidad) {

if (entidad == null)
     throw new NullPointerException("La entidad es nula");

…
}
```

5. Otras cuestiones de implementación

A continuación presentamos información que puede servir de ayuda y debéis tener en cuenta en la realización de la práctica.

5.1 Reflexión

En el momento en que se introdujeron nuevas entidades en el juego se produjeron dos situaciones que van en contra de los principios de extensibilidad y reutilización de la orientación a objetos. Por un lado, el **método `cargarEscenario` de la clase Escenario** debe hacer un **análisis por casos** del tipo de entidad del mapa de propiedades para llamar al método `crear` apropiado. Por otro lado, los métodos `static` **`crear`** se repiten por todas las clases efectivas compartiendo código en común. Se propone utilizar la reflexión para evitar el análisis por casos del método cargar.

Asimismo, se debe ofrecer una alternativa a los métodos crear, que construyen una entidad a partir de un mapa de propiedades, que permita la reutilización del proceso de creación a lo largo de la jerarquía de herencia. Nótese que el criterio que motivó el método crear fue disponer de un método que construyese objetos, pero que fuera capaz de informar un error devolviendo un valor nulo en caso de que el mapa de propiedades fuera incorrecto.

5.2 Enumerados y serialización

```
public class Accion implements Serializable{
  private final transient String nombre;

  public static final Accion DERECHA = new Accion ("Derecha");
  public static final Accion IZQUIERDA = new Accion ("Izquierda");
  public static final Accion SALTA = new Accion ("Salta");
  public static final Accion DISPARA = new Accion ("Dispara");

  private Accion(String nombre){ this.nombre = nombre;}

  public String toString() {return nombre;}
```

```
//Añadir para la serialización
  private static int nextCodigo = 0;
  private final int codigo = nextCodigo++;
  private static final Accion [] VALORES =
     {DERECHA, IZQUIERDA, SALTA, DISPARA};

  private Object readResolve() throws ObjectStreamException{
     return VALORES[codigo];
  }

}
```

5.3 Uso de librerías en JBuilder

Para poder utilizar la librería que contiene las clases de la interfaz gráfica hay que especificar en las propiedades del proyecto dónde se encuentra el fichero `vista.jar`

1) En el menú `Proyecto`, seleccionamos la opción `Propiedades de Proyecto`. En la ventana que se abre tenemos que seleccionar la pestaña `Bibliotecas necesarias`.

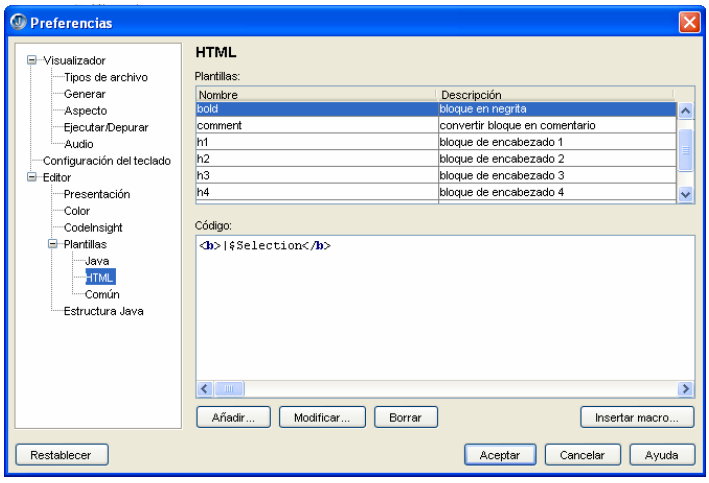
2) Dando al botón `Abrir` se abre una ventana para seleccionar la biblioteca. Debemos pulsar el botón `Nuevo` para especificar la ubicación de la biblioteca con la interfaz gráfica.

3) En el asistente que se abre debemos especificar el nombre que queremos darle a la biblioteca y la ubicación dándole al botón `Añadir`.

5.4 Definición de plantillas de documentación

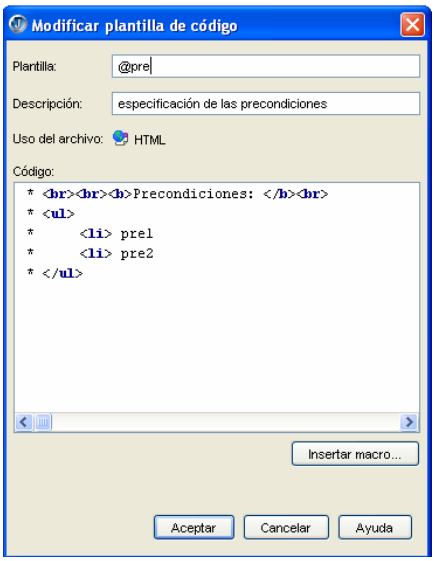
En *JBuilder* podemos definir una plantilla de documentación que nos permita documentar las precondiciones de los métodos implementados. Nótese que también pueden definirse plantillas de código que resultan muy útiles para la programación. A continuación se explica cómo definir una plantilla de documentación:

1) Acceder al menú `Herramientas` y dentro de este a la opción de menú `Preferencias`. Aparecerá la siguiente ventana:

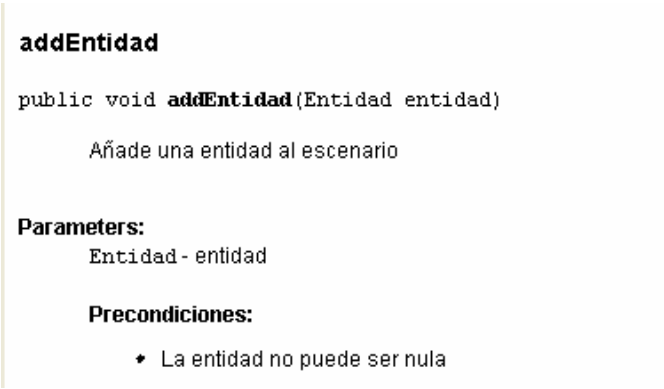


Seleccionar en el panel de la izquierda: `Editor > Plantillas > HTML`

2) Pulsar el botón Añadir … Rellenar la plantilla y pulsar Aceptar.



3) A partir de ahora cada vez que estemos dentro de un comentario de documentación y queramos hacer uso de la plantilla definida no hay más que pulsar ctrl.+J y seleccionar la plantilla @pre. La documentación generada con el *javadoc* quedaría como sigue:



addEntidad

`public void` **addEntidad**`(Entidad entidad)`

Añade una entidad al escenario

Parameters:

`Entidad` - entidad

Precondiciones:

- La entidad no puede ser nula

Plazo y forma de entrega

- El plazo de entrega de la práctica finaliza el ***viernes 16 de junio*** (para la convocatoria de junio) y el ***viernes 8 de septiembre*** (para la convocatoria de septiembre).

- Se debe generar un recopilatorio (fichero .jar) que ha de incluir el código de la aplicación. El recopilatorio debe ejecutarse con el comando: **`java -jar videojuego.jar`**. Por último, la generación del recopilatorio debe realizarse desde proyecto JBuilder, de manera que pueda actualizarse el recopilatorio cada vez que compila la aplicación.

- La práctica se entregará **en SUMA y** también una copia **en papel** del código fuente de la aplicación ordenando las clases de acuerdo a la jerarquía de herencia (no por orden alfabético).