Práctica 4: El lobo y las ovejas

Fecha de entrega: 13 de Marzo de 2017, 9:00

Objetivo: Diseño OO, Contenedores estándar, Genéricos

En las prácticas anteriores hemos construido un intérprete de programas bytecode desde cero. En las prácticas del segundo cuatrimestre trabajaremos sobre un diseño existente, ampliándolo para darle nueva funcionalidad. Este segundo escenario es mucho más habitual en la práctica: muy pocas veces se empieza *ex novo*.

Descripción de la práctica

En esta práctica, el objetivo es hacer una primera ampliación del diseño:

1. Ahora sólo se puede jugar al *Tres en Raya*; añade un nuevo juego "Lobo y Ovejas" y la posibilidad de lanzarlo desde la linea de comandos.
2. Ahora sólo hay un main de ejemplo, que permite jugar al Tres en Raya contra un jugador aleatorio; añade la posibilidad de elegir tanto juego como jugadores desde la línea de comandos.
3. *Opcional*: Implementa pruebas unitarias para verificar que tu nuevo juego funciona como debe; y para comprobar que, efectivamente, interpretas bien los argumentos del main.

IMPORTANTE: No está permitido modificar el código de ninguna clase ó interfaz del paquete es.ucm.fdi.tp.base, o de sus subpaquetes.

Código entregado

El código del que se parte, y que en general no se puede modificar, está estructurado en los siguientes paquetes:

- es.ucm.fdi.tp.base.model: clases base para implementar juegos. Se entiende que todo juego implementado partirá de esas clases, que se describen a continuación:

  - GameState: (abstracta) representa el estado completo de un juego, incluyendo todo lo necesario para poder seguir jugando en otro momento. Por ejemplo, en un juego de tablero, incluirá la posición de las piezas, a quién le toca mover, y cualquier otra información que sea importante para recuperar el estado más tarde. Los estados también saben cómo generar GameActions, que vienen a ser las posibles jugadas. Una vez creado, un estado no se puede modificar (es *inmutable*); esto tiene la ventaja de que pasar estados de un método a otro es seguro, y no requiere hacer copias defensivas.
  - GameAction: (interfaz) describe cómo pasar de un estado a otro, aunque no contiene el estado destino. Por ejemplo, "poner una X en la casilla 1,1" podría ser una acción válida para un estado dado del Tres en Raya; tras aplicarla, devolvería un estado distinto (con una X en esa casilla).
  - GamePlayer: (interfaz) describe lo mínimo que debe hacer un jugador, a saber: decir su nombre, pensar jugadas GameAction, etc.
  - GameError (clase) hereda de RunTimeException y se usa para lanzar excepciones cuando ocurra algún error, etc.

- es.ucm.fdi.tp.base.player: clases con jugadores predefinidos. El RandomPlayer juega al azar, mientras que el SmartPlayer juega de forma inteligente, usando la función de evaluación de las jugadas.

- es.ucm.fdi.tp.base.demo: incluye un ejemplo de main para lanzar un juego.

- es.ucm.fdi.tp.ttt: clases para implementar el juego del *Tres en raya*. Se juega sobre un tablero.

Durante las clases y sesiones de laboratorio se proporcionará más detalles sobre el código entregado.

Maven

Las prácticas 4, 5 y 6 usarán Maven[1] para describir cómo se compilan, prueban, ejecutan y empaquetan los proyectos. Maven es un estándar en el mundo de Java[2], similar a NPM para Javascript/NodeJS, Packagist para PHP, ó Rubygems para Ruby.

Para importar el proyecto en Eclipse, se debe seleccionar Import >Existing Maven Project; y a continuación se debe navegar hasta la carpeta que contiene el archivo pom.xml. La estructura de un proyecto Java que usa Maven es la siguiente:

- pom.xml: describe el proyecto, incluyendo sus dependencias y todo su ciclo de vida (validación, compilación, pruebas, ejecución, y empaquetado). Recomendamos no modificarlo sin consultar al profesor.

- src/main/java/: contiene los fuentes de Java. En un proyecto Eclipse estándar, estarían bajo src/.

- src/main/resources/: incluye recursos (imágenes, ficheros de configuración o localización, etcétera) que se incluirán en el paquete final, pero que no son código fuente.

[1] ver https://maven.apache.org/what-is-maven.html para una introducción

[2] hay más de 175000 paquetes Maven disponibles, y casi todos los proyectos Java en Github incluyen un pom.xml, es decir, usan Maven

- src/test/java/: contiene fuentes de java que se usan exclusivamente en la fase de pruebas. Pediremos que implementes pruebas para verificar que tu código funciona.
- target/: creado por Maven para contener todos los resultados de compilar y probar el código. Se puede borrar sin problemas (se regenera cada vez que es necesario). El equivalente en un proyecto Eclipse estándar es bin/

Una vez importado en Eclipse, éste creará sus tradicionales ficheros .project, .classpath y el directorio .settings/. Si los borras, tendrás que re-importar el proyecto para poder volver a abrirlo.

Pruebas con JUnit

Las pruebas unitarias son una forma de verificar que los métodos de las clases funcionan como se espera. Se llaman unitarias porque prueban las unidades más pequeñas que tiene sentido probar en aislamiento unas de otras (en oposción a las pruebas de integración, que verifican que todo funciona una vez juntado). Usaremos la librería JUnit, que cuenta con soporte tanto Maven como Eclipse, para escribir y ejecutar pruebas unitarias en nuestras prácticas.

Para pasar las pruebas unitarias bajo Eclipse[3], selecciona la raiz del proyecto en la vista de paquetes, haz click derecho en el mismo, y usa la opción Run As ... >JUnit Test. Esto también funciona si seleccionas cualquier rama del proyecto que contenga pruebas (por ejemplo, src/test/java/).

Para crear una nueva prueba unitaria con Eclipse, selecciona la clase a probar en la vista del explorador de paquetes, haz click derecho en la misma, y usa la opción New >JUnit Test Case . A continuación, cambia el lugar donde se creará la clase de *tu-proyecto*/src/main/java a *tu-proyecto*/src/test/java, dejando las demás opciónes en sus valores por defecto.

Cuando JUnit pasa las pruebas, sencillamente va ejecutando todos los métodos anotados con @Test. Si el método no llama a fail, no contiene ningún assert que falle, y no lanza excepciones incontroladas, se considera que la prueba ha tenido éxito. En caso contrario, se considera que ha fallado.

Ejemplo de pruebas con JUnit:

```
@Test
public void pruebaSumasSencillas() {
    Calculadora c = new Calculadora();
    // argumentos: mensaje, esperado, obtenido
    assertEquals("uno mas uno dos", 1+1, c.suma(1,1));
    assertEquals("dos mas dos cuatro", 4, c.suma(2,2));
}

@Test
public void pruebaDivisionPorCero() {
    Calculadora c = new Calculadora();
    try {
        c.divide(1, 0);
        fail("uno entre cero debe fallar");
    } catch (DivisionPorCeroException dpce) {
```

[3]Una de las ventajas de Maven es que también puedes pasar pruebas *sin* Eclipse - escribiendo mvn test en la consola, por ejemplo.

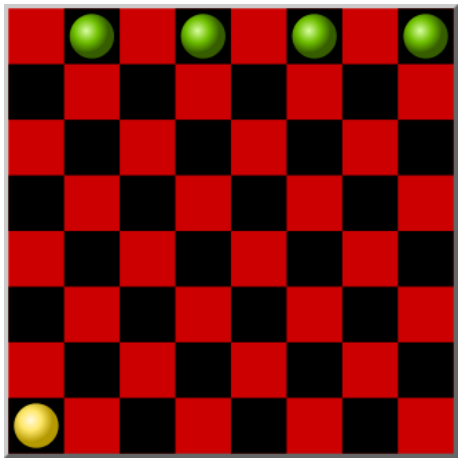

Figura 1: Tablero del juego del Lobo (abajo) y las Ovejas (arriba)

```
            // todo bien
        }
}
```

Entrega

Debes entregar un único zip (y no .rar, ni .7z, ni .tgz: un **.zip**) por grupo vía campus virtual, conteniendo exclusivamente los siguientes archivos y directorios: pom.xml, src/ (y todo lo que hay debajo), y un fichero llamado alumnos.txt. En alumnos.txt debes incluir el nombre de integrantes del grupo, así como cualquier comentario que quieras que tu profesor tenga en cuenta durante la corrección de la práctica.

Enunciado detallado

A continuación se proporcionan más detalles acerca de los ejercicios de los que se compone la práctica.

El Lobo y las Ovejas

Este juego requiere un tablero de 8×8 similar al usado para el ajedrez, donde sólo se usarán las casillas negras (ver ilustración). Las ovejas se sitúan en las casillas negras de un extremo, mientras que el lobo ocupa la esquina opuesta.

El juego transcurre por turnos, empezando a mover el jugador que controla al lobo. El lobo puede moverse, en diagonal, 1 casilla en cualquier dirección. Las ovejas pueden moverse, también en diagonal, 1 casilla; pero sólo avanzando hacia el lado opuesto a su posición inicial. El lobo gana si consigue alcanzar el extremo del tablero en el que empezaban las ovejas. Las ovejas ganan si consiguen que el lobo quede inmovilizado. Se supone que el jugador 0 es el que controla el lobo, y el jugador 1 es el que controla la ovejas.

Para implementar este juego, deberás

1. crear un paquete es.ucm.fdi.tp.was

2. añadir, en este paquete, una clase WolfAndSheepState que extienda de GameState; y que permita crear jugadas (ver punto siguiente)

3. añadir, en este paquete, una clase WolfAndSheepAction que implemente GameAction (y que permita crear nuevos estados al aplicarla a un WolfAndSheepState).

Nuevo main

El main de prueba (es.ucm.fdi.tp.demo.Main) contiene ejemplos que lanzan juegos y jugadores aleatorios, automáticos y de consola. No obstante, el main no acepta argumentos, y siempre hace lo mismo.

Para este apartado, vas a escribir un main mejorado que permita, en función de los parámetros de consola recibidos, lanzar cualquiera de los 3 juegos disponibles, tanto con jugadores "por consola" como con jugadores aleatorios ó inteligentes.

Sintaxis: *juego* $jugador_1$ $jugador_2$, donde

- juego: ttt (para el Tres en Raya) ó was (para Wolf and Sheep).
- jugador: console (para que ese jugador se controle por consola), rand (para que sea un jugador aleatorio) ó smart (para que sea un jugador inteligente).

Si algún argumento no es válido, debes mostrar un error que lo explique y finalizar la ejecución. Por ejemplo, si intento jugar al juego chess (que no está definido), debes mostrar un mensaje que indique que ese juego no es válido.

Ejemplos:

parámetros	resultado
`ttt console rand`	Tres en Raya, con un jugador de consola contra uno aleatorio
`ttt luis pedro`	Error: jugador "luis" no definido
`ttt console rand console`	Error: demasiados jugadores para este juego

Para implementar esta funcionalidad, recomendamos

- crear un nuevo paquete es.ucm.fdi.tp.launcher.
- crear una clase es.ucm.fdi.tp.launcher.Main.
- crear un método estático en esta clase con la firma

  ```
  public static GameState<?,?> createInitialState(String gameName)
  ```

  donde gameName podrá ser cualquiera de las abreviaturas de juegos anteriormente descritas.

- crear un método estático en esta clase con firma

  ```
  public static GamePlayer createPlayer(String gameName,
      String playerType, String playerName)
  ```

donde gameName es como antes, y playerType podrá ser cualquiera de los tipos de jugador anteriormente descritos.

- copiar el método playGame de es.ucm.fdi.tp.demo.Main al nuevo main.
- crear un método main que interprete los argumentos (con la ayuda de createInitialState y createPlayer) y muestre posibles mensajes de error. Si no se produce ningún error, llama a playGame para empezar a jugar. Usa una lista de nombres predefinida para los nombres de jugadores.

Pruebas unitarias (Opcional)

Escribe pruebas unitarias para verificar que los apartados 1 y 2 funcionan correctamente. Para ello, sigue las instrucciones del apartado *Pruebas con JUnit* para:

- crear una clase es.ucm.fdi.tp.was.WolfAndSheepStateTest bajo *tu-proyecto*/src/test/java en la que se compruebe, al menos, que
  - un lobo rodeado resulta en victoria de las ovejas
  - un lobo en una casilla con $y = 0$ resulta en victoria del lobo
  - un lobo en su posición inicial sólo tiene 1 acción válida; y tras llevarla a cabo, en su siguiente turno, tiene 4 acciones válidas.
  - una oveja en su posición inicial tiene 2 acciones válidas; y si está en un lateral, tiene 2 acciones válidas.
- crea una clase es.ucm.fdi.tp.launcher.MainTest bajo *tu-proyecto*/src/test/java en la que se compruebe, al menos, que
  - proporcionar menos de 3 argumentos ó demasiados argumentos (más jugadores de los que acepta el juego) resulta en un error.
  - proporcionar un juego inválido como primer argumento resulta en un error.