Prácticas de diseño de patrones utilizando la familia de juegos del Conecta 4

Marco Antonio Gómez Martín
Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Noviembre 2009

Documento maquetado con TEXIS v.1.0.

Prácticas de diseño de patrones utilizando la familia de juegos del Conecta 4

Autor:
Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Noviembre 2009

ISBN 978-84-692-7108-7

Resumen

Este Manual Docente contiene una secuenciación de prácticas de Java basadas en la familia de juegos "*n-in-a-row*" a los que pertenece entre otros el Conecta 4 y las 3 en raya.

Cada práctica se construye en base al desarrollo de la práctica anterior, de forma que la última práctica *subsume* el comportamiento de todas las anteriores. De esta forma, el desarrollo de una práctica no consiste únicamente en la creación de una serie de clases, sino la *reutilización* de código desarrollado previamente. Este modo de actuar permite poner en práctica conceptos de orientación a objetos en general y de patrones de diseño en particular. De esta forma, al ir avanzando en las prácticas se pone de manifiesto la necesidad (o conveniencia) de utilizar distintos patrones de diseño.

Las secuencia de prácticas fue utilizada en la asignatura de *Laboratorio de Programación de Sistemas* durante el curso 2007/2008 en la Facultad de Informática de la Universidad Complutense de Madrid. La asignatura pertenece al tercer curso de la titulación de *Ingeniería en Informática de Sistemas*. El manual docente conserva incluso el pie de página utilizado en el cuadernillo proporcionado a los alumnos. También hemos creido conveniente dejar la *fecha de entrega* de cada una de las prácticas. De esta forma el lector podrá hacerse una idea de la cantidad de tiempo que tuvieron los alumnos para realizar cada una de ellas.

La experiencia con estas prácticas ha dado lugar a dos publicaciones en congresos relacionados con la enseñanza de la informática:

Gómez-Martín, M.A., Jiménez-Diaz, G. y Arroyo-Gallardo, J. *eaching Design Patterns Using a Family of Games*. 14th ACM-SIGCSE Annual Conference on Innovation and Technology in Computer Science. ACM Press. 2009.

Gómez-Martín, M.A. y Gómez-Martín, P.P. *Fighting against the 'But it works!'syndrome*. XI International Symposium on Computers in Education. 2009.

Índice

Índice de figuras

Práctica 0: Iniciación a Java

Fecha de entrega: No debe entregarse

Esta práctica consiste en una iniciación a Java y a las herramientas que se utilizarán durante el curso.

Es una práctica guiada donde se deben ir realizando los pasos indicados.

Material proporcionado:

Fichero	Explicación
`junit-4.4.jar`	Librería de tests.
`build.xml`	Fichero utilizado en apartado 6.

Importante: Para el correcto funcionamiento de los comandos indicados, se asume que el `PATH` del sistema está bien configurado para que encuentre las aplicaciones de Java necesarias (compilador, máquina virtual, etc.).

1. Primer programa

1.1. Escribiendo el código

El primer paso es crear un primer programa sencillo en Java y ejecutarlo *desde la línea de comandos*. Para ello, crear el fichero `Main.java` con cualquier editor (Bloc de notas, emacs...) con el siguiente contenido:

```
/**
 * Clase principal de la práctica 0.
 */
public class Main {
```

```
  /**
   * Método que devuelve una cadena de saludo.
   * @return Devuelve la cadena "Hola mundo"
   */
  public String saludo() {
    return "Hola mundo";
  }

  /**
   * Método que devuelve una cadena de despedida.
   * @return Devuelve la cadena "Adios mundo cruel"
   */
  public String despedida() {
    return "Adios mundo cruel";
  }

  /**
   * Función de entrada a la aplicación.
   * @param args Argumentos de la aplicación.
   */
  public static void main(String args[]) {
    Main m = new Main();

    System.out.println(m.saludo());
  }
}
```

1.2. Compilando el código

Para compilar el código anterior, se utiliza el compilador de Java, `javac`. Desde la línea de comandos (Inicio >Programas >Accesorios >Símbolo del sistema), ejecutar:

```
javac Main.java
```

Eso creará un fichero nuevo, `Main.class`, que contiene la aplicación compilada[1].

1.3. Ejecutando la aplicación

Para ejecutarla, se invoca `java`, el programa que implementa la máquina virtual de Java y que recibe como parámetro la clase principal (la que contiene el método `main`):

```
java Main
```

[1] Si no encuentra el comando `javac`, puede que no hayas leído la nota **Importante** del principio del enunciado.

Si todo ha ido bien, se habrá escrito la cadena "`Hola mundo`" por la consola.

1.4. Generación de la documentación

El código anterior incluía comentarios utilizando el formato de *javadoc*, de tal forma que podemos utilizar la herramienta para generar los HTML con la documentación de la clase.

```
javadoc Main.java -d doc
```

La opción "`-d doc`" hace que los ficheros de documentación se generen en el directorio `doc`.

Se puede ver el resultado en `./doc/index.html`.

2. Primer test

Para garantizar el correcto funcionamiento de las clases programadas, debemos acostumbrarnos a implementar *test de unidad*, que comprueban que las funciones implementadas hacen lo que deben.

Aunque en este caso la clase es muy sencilla, vamos a crear una clase de test que comprueba que los métodos de la clase `Main` son correctos. Para la realización de los tests utilizaremos la librería JUnit.

2.1. Código del test

Para probar la clase `Main`, escribimos una nueva clase, `MainTest`, que debe heredar de la clase `TestCase` de JUnit, y que tiene dos métodos cuyos nombres empiezan por `test` (lo obliga JUnit): `testSaludo` y `testDespide`. Cada uno comprueba que el método correspondiente funciona.

```
import junit.framework.TestCase;

public class MainTest extends TestCase {

  public void testSaludo() {
    Main m = new Main();

    assertEquals("Saludo falla", m.saludo(), "Hola mundo");
  }

  public void testDespedida() {
    Main m = new Main();

    assertEquals("Despedida falla", m.despedida(),
                 "Adios mundo cruel");
```

```
  }
}
```

2.2. Compilando el test

Para compilar el test se utiliza, igual que antes, `javac`:

```
javac MainTest.java
```

Sin embargo, ahora la compilación *falla*, debido a que el compilador no encuentra la librería JUnit (necesita la clase `junit.framework.TestCase` de JUnit, y no encuentra ni el código para compilarla ni la clase ya compilada en el CLASSPATH):

```
MainTest.java:1: package junit.framework does not exist
import junit.framework.TestCase;
                       ^
MainTest.java:3: cannot find symbol
symbol: class TestCase
public class MainTest extends TestCase {
                              ^
MainTest.java:8: cannot find symbol
symbol  : method assertEquals(java.lang.String,java.lang.String,
java.lang.String
)
location: class MainTest
        assertEquals("Saludo falla", m.saludo(), "Hola mundo");
        ^
MainTest.java:14: cannot find symbol
symbol  : method assertEquals(java.lang.String,java.lang.String,
java.lang.String
)
location: class MainTest
        assertEquals("Despedida falla", m.despedida(), "Adios");
        ^
4 errors
```

Debemos indicar al compilador *dónde* puede encontrar la librería JUnit compilada. Ésta se encuentra en un fichero con extensión `.jar`. En la invocación al compilador, indicamos que puede buscar las clases en ese `.jar`, mediante el parámetro `-cp`[2]:

[2]Debemos copiar el fichero `junit-4.4.jar` al directorio actual.

```
javac -cp "junit-4.4.jar" MainTest.java
```

Al compilar, vemos que aparece otro error: no encuentra la clase `Main` (se necesita porque se crean objetos de la misma). El error se da porque *no* hemos dicho al compilador que busque las clases que necesite en el directorio actual ("`.`").

Finalmente:

```
javac -cp "junit-4.4.jar;." MainTest.java
```

2.3. Ejecutando los test

Para ejecutar los tests, se utiliza la clase `junit.textui.TestRunner`, que recibe como parámetro el nombre de la clase que implementa los test. Para que encuentre la clase, hay que indicar en el parámetro `-cp` el `.jar`:

```
java -cp "junit-4.4.jar;." junit.textui.TestRunner MainTest
```

El resultado es:

```
..
Time: 0

OK (2 tests)
```

Se puede cambiar alguno de los tests, para que falle (comparando con una cadena distinta).

3. Ordenando el directorio

El directorio donde hemos trabajado en el apartado anterior, debe tener:

- `Main.java`: código fuente de la clase.
- `Main.class`: clase `Main` compilada.
- `MainTest.java`: código fuente de la clase de prueba.
- `MainTest.class`: clase `MainTest` compilada.
- `doc`: directorio con la documentación.

Cuando el programa Java crece, el número de ficheros también, por lo que es interesante mantener un orden. En general, *siempre* organizaremos el código de tal forma que:

- El código de las clases se encuentre en el directorio `src`.

- Las clases de *test* se almacenan en el directorio `tests`.
- Las clases compiladas se guarden en `bin` (en ocasiones, también se utiliza el directorio `classes`).

Haciéndolo así, mantenemos cada tipo de fichero en un directorio distinto.

Creamos los directorios anteriores, `src`, `tests` y `bin`. Movemos `Main.java` al directorio de código fuente, y `MainTest` al directorio `tests`. Los ficheros `.class` los borramos.

Para compilar la aplicación, hay que indicar al compilador `javac` que el código fuente está en el directorio `./src`, y que queremos que deje los ficheros compilados en `./bin`:

```
javac -d "bin" src/Main.java
```

Para ejecutar la aplicación, hay que indicar dónde están los ficheros compilados.

```
java -cp "./bin" Main
```

Para compilar los tests, se procede de igual forma que antes. Lo único que cambia es el sitio en el que debe buscar la clase `Main` compilada:

```
javac -cp "junit-4.4.jar;./bin" -d "bin" tests/MainTest.java
```

Y para ejecutar el test, de forma similar:

```
java -cp "junit-4.4.jar;./bin" junit.textui.TestRunner MainTest
```

4. Usando paquetes Java

El siguiente paso para mejorar el programa es hacer uso de paquetes (`package`). Vamos a meter la clase `Main` en el paquete `lps.pr0`. Para eso, en el directorio `src`, debemos crear el directorio `lps`, y dentro de éste `pr0`. En `src/lps/pr0` es donde irá ahora la clase `Main`[3].

Para compilar, se utiliza:

```
javac -d "bin" src/lps/pr0/Main.java
```

Observa que en el directorio `bin` se ha creado la misma estructura de directorios, es decir, la clase compilada se crea en `bin/lps/pr0`.

Mueve la clase de tests al paquete `lps.pr0.tests` y compilala[4].

[3] No olvidar poner la instrucción `package lps.pr0;` en la primera línea del fichero.

[4] Observa que la clase `Main` que utilizan los tests ya no está en el paquete por defecto o raiz, por lo que tendrás que modificar el código.

5. Agrupando la aplicación

Se pueden agrupar todas las clases (compiladas en el directorio `bin`) en un `.jar`, para poder distribuirlo.

Para crear el `.jar`, se utiliza la herramienta `jar`, incluida en la distribución de Java.

Desde el directorio `./bin` (donde están las clases compiladas):

```
jar cvf ../practica0.jar .
```

Que crea (opción `c`) un nuevo fichero de empaquetado, cuyo nombre damos (`f`), llamado `../practica0.jar` (para que lo cree en el directorio padre). Durante la creación, da detalles de las acciones que hace (`v`), entre otras cosas de la compresión conseguida de cada fichero. En el archivo añade todo el directorio (`.`).

El fichero puede abrirse con programas como WinZip o WinRar. Se observa que mantiene la estructura de directorios, y contiene los ficheros `.class`.

Una vez que tenemos el fichero empaquetado, podemos ejecutar la aplicación directamente desde él. Si vamos al directorio donde se encuentra el jar:

```
java -cp practica0.jar lps.pr0.Main
```

Por último, podemos añadir información al `.jar` que indique cuál es la clase principal de la aplicación.

Para eso, creamos un fichero de texto (`manifest.mf`) que contenga la información:

```
Manifest-Version: 1.0
Main-Class: lps.pr0.Main
```

Entonces, ejecutamos la orden `jar` anterior, pero indicando además el archivo "*de manifiesto*" que queremos que sea añadido. Si hemos creado el fichero en el "raiz", y estamos en el directorio `./bin`:

```
jar cvfm ../practica0.jar ../manifest.mf .
```

La diferencia entre ambos, radica en la opción `m`, que indica que se quiere incluir un archivo de manifiesto dado.

Con el `.jar` así creado, se puede ejecutar la aplicación directamente, sin necesidad de especificar la clase que contiene el `main`:

```
java -jar practica0.jar
```

6. Ant, el make de Java

En general, la compilación, creación del `.jar`, documentación y ejecución de los tests desde la línea de comandos es tediosa. Para facilitar la tarea, se puede utilizar un archivo por lotes (`.bat` en Windows), o la herramienta *make*.

Sin embargo, para Java, existe una herramienta mucho más conveniente, llamada Ant[5]. Ésta lee de un fichero XML las instrucciones para la generación del proyecto Java, y las ejecuta.

Por defecto, se puede ejecutar la utilidad `ant`, y éste leera el fichero `build.xml`. Todas las tareas realizadas de forma manual en los apartados anteriores son realizados utilizando el siguiente fichero:

```
<?xml version="1.0"?>
<!-- Fichero Ant para la practica 0. La ausencia de -->
<!-- acentos en los comentarios NO es casualidad.   -->

<project name="Practica0" default="all" basedir=".">

<!-- Definicion de propiedades (= variables dentro  -->
<!-- del fichero).                                  -->
<property name="srcDir" value="src"/>
<property name="testDir" value="tests"/>
<property name="buildDir" value="bin"/>
<property name="docDir" value="doc"/>

<!--                 -->
<!--   COMPILACION   -->
<!--                 -->
<target name="compile" description="Compilacion">
  <javac srcdir="${srcDir}" destdir="${buildDir}"/>
  <jar destfile="pr0.jar" basedir="${buildDir}">
    <manifest>
      <attribute name="Main-Class" value="lps.pr0.Main"/>
      <attribute name="Built-by" value="${user.name}"/>
    </manifest>
  </jar>
  <echo>Compilacion completa</echo>
</target>

<!--                 -->
<!-- DOCUMENTACION   -->
<!--                 -->
<target name="documentacion"
        description="Generacion de documentacion">
   <javadoc destdir="${docDir}"
            sourcepath="${srcDir}"
```

[5] `http://ant.apache.org`

```
            author="true"
            windowtitle="Practica 0"/>
   <echo>Documentacion generada</echo>
</target>

<!--                                -->
<!--   COMPILACION DE LOS TEST      -->
<!--                                -->
<target name="compileTests" description="Compilacion de tests">
   <javac srcdir="${testDir}" destdir="${buildDir}"
          classpath="junit-4.4.jar"/>
   <echo>Compilacion de tests completa</echo>
</target>

<!--                                -->
<!--   EJECUCION DE LOS TEST        -->
<!--                                -->
<target name="runTests"
        description="Ejecucion de tests">
   <junit printsummary="yes">
      <classpath>
         <pathelement path="${buildDir}"/>
    <pathelement location="junit-4.4.jar"/>
      </classpath>
      <formatter type="plain"/>
      <test name="lps.pr0.tests.MainTest"/>
   </junit>
</target>

<!--                                -->
<!--   OBJETIVO COMPLETO            -->
<!--                                -->
<target name="all"
        depends="compile, documentacion, compileTests, runTests"
        description="Construye el proyecto completo">
  <echo>Terminado</echo>
</target>

</project>
```

Si todos los pasos anteriores se han hecho correctamente, la ejecución del comando `ant` compilará la clase `Main`, compilará el test y lo ejecutará. También genera el `.jar` y genera la documentación.

Práctica 1: Laberinto

Fecha de entrega: 29 de Noviembre de 2007

Material proporcionado:

Fichero	Explicación
`testPr1.jar`	Tests de algunas clases de la práctica que deben ejecutarse satisfactoriamente.
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega. También puede utilizarse su objetivo `runTests` para ejecutar todos los tests anteriores sobre la práctica.
`laberinto.txt`	Ejemplo de un laberinto.

Importante: Se aconseja que se vayan ejecutando los tests proporcionados, para comprobar si se está implementando la práctica correctamente. Ver apartado 4 para más detalles.

1. Descripción

Se trata de implementar una aplicación que muestre un laberinto en modo texto y permita al usuario jugar para buscar la salida del mismo.

Para ello, el programa le irá preguntando la dirección en la que quiere moverse, y dibujará el laberinto con el personaje en la nueva posición.

En esta práctica, por ser la primera, explicaremos con detalle **todas** las clases que se deben implementar, junto con sus métodos. Se aconseja seguir el orden indicado, e ir creando test de unidad para probar su correcto funcionamiento, aunque en esta práctica no es obligatorio hacerlo.

En la explicación que sigue se indica, para cada clase, los tests involucrados que la clase deberá pasar. Ver el apartado 4 para más detalles.

La práctica consta de dos partes, ambas obligatorias.

2. Parte 1

Clase de test: `lps.pr1.testprofesor.TestParte1`

Todas las clases de la aplicación deben estar en el paquete `lps.pr1`. Dentro de éste, existe una única clase, `MainParte1`, que contiene el punto de entrada de la aplicación.

Además, existen tres paquetes:

- `logica`: Contiene la parte de la "lógica" del laberinto. En particular, tiene las clases que representan una dirección, localización y casilla del laberinto. También tiene una clase para almacenar el laberinto, así como una que representa la partida.
- `gui`: En este paquete aparecen las clases encargadas de la representación del laberinto en pantalla. En esta práctica nos conformaremos con dibujar el laberinto en la consola, utilizando los métodos `print` y `println` de `System.out`.
- `aplicacion`: Contiene la clase que controla el flujo de ejecución de la aplicación.

En las siguientes secciones aparece una descripción detallada de cada paquete y las clases que deben contener.

2.1. Paquete `logica`

2.1.1. Direccion

Representa una dirección dentro del laberinto. Es un simple enumerado con cuatro valores, uno por cada punto cardinal.

Clase de test: `lps.pr1.logica.testprofesor.DireccionTestAPI`

2.1.2. Casilla

Representa el tipo de casilla del laberinto dentro de la matriz bidimensional. Es un enumerado con dos valores, uno para representar que existe `Muro` y otro para indicar que la casilla es un pasillo o `Camino`.

Clase de test: `lps.pr1.logica.testprofesor.CasillaTestAPI`

2.1.3. Localizacion

Representa un punto dentro del laberinto, por lo que tiene dos atributos para almacenar las coordenadas `x` e `y`. Tiene, al menos, los siguientes métodos públicos:

- Constructores: dos, uno sin parámetros y otro con dos parámetros para indicar la posición inicial.
- Métodos de acceso: `getX` y `getY`, para saber las coordenadas.
- Métodos de modificación: `setX` y `setY`, para cambiarlas.
- `avanza`: recibe como parámetro una `Direccion`, y devuelve la localización resultante de moverse en esa dirección desde esa posición.
- `clone` y `equals`
- `readFromBufferedReader`: recibe un `java.io.BufferedReader` y lee de la siguiente línea las dos coordenadas enteras (primero la posición x y luego la y). Si hay algún problema en la lectura o al convertir a enteros los datos leídos, genera la excepción `java.io.IOException`.

Clase de test: `lps.pr1.logica.testsprofesor.LocalizacionTest`

2.1.4. MatrizLaberinto

Esta clase contiene los datos de un laberinto. Desde el punto de vista *lógico*, un laberinto es una matriz bidimensional de celdas con un ancho y un alto. Cada celda puede ser o bien muro o bien camino.

Para acceder a los datos que almacena, deberá tener *al menos* los siguientes métodos públicos:

- Un constructor con dos parámetros enteros, el primero de ellos indicando el ancho, y el segundo el alto. El laberinto estará formado por muros en todas las casillas.
- `getAncho`: devuelve un entero con el ancho de la matriz.
- `getAlto`: devuelve un entero con el alto de la matriz.
- `ponCamino`: hace que la posición que recibe como parámetro se convierta en camino; si ya lo era o la posición es inválida, no hace nada. Hay *dos* versiones del método, una recibiendo una pareja de enteros (x, y) (entre 0 y n-1), y otra que recibe una `Localizacion`.
- `ponMuro`: cancela el efecto del método anterior.

- `dameCasilla`: dada una pareja de enteros (ancho, alto), devuelve una `Casilla` indicando lo que hay en esa posición. Si la posición no es válida, devuelve un `Muro`.
- `equals`

Deberá además implementar el método `readFromBufferedReader`, cuya especificación ("cabecera") es igual que para la clase `Localizacion`. El formato esperado es el siguiente[1]:

- La primera línea contiene el número de columnas (ancho) del laberinto.
- La segunda línea contiene el número de filas (alto) del laberinto.
- La tercera línea contiene el número uno ("`1`"). Esto es así por compatibilidad con un formato de datos anterior[2].
- La cuarta línea contiene el número de casillas del laberinto que *no* son muro.
- A continuación aparece una línea por cada casilla de tipo camino. Cada línea contiene la posición de esa casilla (la esquina superior izquierda es la posición (0, 0)).

Recuerda que si hay algún error de formato o de lectura, el método debe generar una excepción del tipo `java.io.IOException`.
Clase de test: `lps.pr1.logica.testsprofesor.MatrizLaberintoTest`

2.1.5. Partida

La clase contiene información de una partida; en particular, almacena el laberinto (una `MatrizLaberinto`), las posiciones de la entrada y la salida dentro de él, y la posición del jugador. También guarda un objeto del tipo `lps.pr1.gui.LaberintoGUI` para el dibujado (aparece descrita en la siguiente sección).

Debe tener *al menos* los siguientes métodos públicos:

- Un constructor que reciba el objeto con el que se pintará el tablero (del tipo `lps.pr1.gui.LaberintoGUI`).
- Un método llamado `mueveJugador`, que reciba una `Direccion` y mueva al jugador de acuerdo a ella. Si no se pudo mover al jugador, devolverá `false`.

[1] Coincide con el utilizado en la práctica 3 de LP2 del Curso 2006/2007, por lo que podrás reutilizar los ficheros para probarlo.

[2] El de la práctica 2 de LP2 del curso 2006/2007.

- Un método `terminado` que devuelva un valor binario que indica si la partida ha terminado (es decir, si el jugador está sobre la casilla de salida).
- Un método `dibuja`, que pinte el estado de la partida utilizando el objeto del tipo `LaberintoGUI` especificado en el constructor.

La clase debe tener también un método para leer de disco el estado de una partida. Igual que en las clases anteriores, el método se llamará `readFromBufferedReader`, y tendrá la misma declaración. El formato del fichero que lee será el siguiente:

- La primera línea contiene la posición de entrada del caminante/jugador.
- La segunda línea contiene la posición de salida.
- A continuación aparece la información sobre el estado del laberinto, según lo descrito en la descripción de la clase `MatrizLaberinto`.

La función generará la excepción si hay error de lectura o error en el formato.

Clase de test: `lps.pr1.logica.testsprofesor.PartidaTestAPI`

2.2. Paquete `gui`

2.2.1. LaberintoGUI

Es un *interfaz* que implementan todas aquellas clases que son capaces de dibujar laberintos, ya sea en la consola o en una ventana gráfica.

Los métodos que tiene son los siguientes (ninguno de ellos devuelve nada):

- `beginPaint(int, int)`: marca el comienzo del dibujado. Los parámetros indican el ancho y alto de la matriz que hay que dibujar.
- `endPaint()`: marca el final del dibujado.
- `dibujaLaberinto(MatrizLaberinto)`: método para indicar qué laberinto hay que dibujar.
- `dibujaCaminante`, `dibujaEntrada` y `dibujaSalida`: los tres reciben una `Localizacion` indicando la posición concreta.

Los métodos `beginPaint` y `endPaint` aparecen por si la implementación de la clase concreta necesita saber cuándo se termina el dibujado. En algunas implementaciones del interfaz puede que el cuerpo de ambos esté vacio.

Clase de test: `lps.pr1.gui.testsprofesor.LaberintoGUITestAPI`

2.2.2. LaberintoGUIConsola

Esta clase implementa el interfaz anterior, de tal forma que dibuja en la consola el laberinto indicado, utilizando `System.out`.

La forma de dibujar es la siguiente:

- Los muros se pintan con "`X`".
- Los caminos se "pintan" con espacios.
- La entrada se pinta con "`E`".
- La salida se pinta con "`S`".
- El personaje se pinta con "`O`", y debe ser siempre visible.

La clase debe tener, además de los métodos del interfaz, un método `getLastString`, que devuelva la cadena con la que se dibujó el laberinto la última vez.

Ejemplos de laberintos son:

```
XXXX        XXXX        XXXX        XXXX        XXXX
O  X        EO X        E OX        E  X        E  X
XX S        XX S        XX S        XXOS        XX O
```

Clase de test: `lps.pr1.gui.testsprofesor.LaberintoGUIConsolaTest`

2.3. Paquete `aplicacion`

Consta de una única clase, `Aplicacion`, que es la responsable de la ejecución del programa. Desde el método `main` de la clase `lps.pr1.MainParte1` se configurará un objeto `Aplicacion` y se invocará a su método `run` para jugar.

Debe tener *al menos* los siguientes métodos públicos:

- `setPartida(Partida)`: establece el objeto de tipo partida que se debe utilizar para jugar. El objeto en cuestión se habrá creado fuera y ya tendrá establecido el objeto con el que se pintará el laberinto.
- `Direccion pideDireccion()`: pregunta al usuario la nueva dirección hacia la que quiere ir, y la devuelve. El método *debe* garantizar que se puede ir en esa dirección (para eso, es posible que necesites añadir algún método público a la clase `Partida`, para averiguar si una dirección es válida).
- `boolean jugarOtra()`: pregunta al usuario si quiere jugar otra partida, y devuelve la respuesta.

- `void iniciarPartida(Partida)`: inicializa el objeto partida con una partida nueva. En particular, pregunta al usuario el fichero del que quiere cargar la partida. Si la lectura produce algún error, informa al usuario, y vuelve a pedir otro fichero.

- `void run()`: va jugando partidas, invocando a los métodos anteriores. En particular el método invocará `iniciarPartida` al principio de cada partida, utilizando el mismo objeto que se pasó desde el `main` mediante `setPartida`.

Clase de test: `lps.pr1.aplicacion.testsprofesor.AplicacionTest`

3. Parte 2

Clase de test: `lps.pr1.testprofesor.TestParte2`

En la segunda parte, vamos a añadir nuevos métodos a la clase `Partida` y `MatrizLaberinto` para que pueda inicializar un laberinto de forma aleatoria.

Crearemos además una clase nueva que hereda de `Aplicacion` que utiliza la inicialización anterior en vez de pedir el nombre de un fichero. Además, implementa el comportamiento automático del jugador, utilizando la técnica de "siempre a la izquierda" para recorrerlo.

Para probarla, existirá una clase `MainParte2` en el paquete `lps.pr1`, que utiliza la nueva clase aplicación para ejecutar el juego.

Los cambios aparecen descritos a continuación.

3.1. Clase `MatrizLaberinto`

Se añade un nuevo método, `initAleatorio(int, int)`, que inicializa un laberinto de manera aleatoria. Los parámetros indican el tamaño (ancho, alto) del nuevo laberinto.

El método de creación empieza con todas las casillas del laberinto con muro. Se abre un camino en la posición (0, 0), y se procede de la siguiente forma:

- Elegimos una dirección hacia la que podamos ir. Sólo son válidas las direcciones que conducen a casillas muro que tengan una única casilla adyacente con camino. La casilla a la que llegamos la convertimos a camino.

- Si no existe ninguna dirección que cumpla esas condiciones, volvemos hacia atrás de donde veníamos[3].

- El algoritmo termina cuando ya no podemos ir hacia atrás más.

[3] Necesitarás una pila; puedes implementar una, o utilizar `java.util.Stack`.

Para elegir una dirección de forma aleatoria, el método debe llamar a un nuevo método público de la clase, `eligeEntero`, que devuelve un número entre 0 y n-1[4]:

```
/**
 * Elige un número entre 0 y n-1
 * @param n Rango de valores a elegir.
 * @return Número aleatorio entre 0 y n-1
 */
public int eligeEntero(int n) {
   return (int)(Math.random() * n);
}
```

Clase de test: `lps.pr1.logica.testsprofesor.MatrizLaberintoTestAPI2`

3.2. Clase `Partida`

La clase tiene un método nuevo, `initAleatorio(int, int)` que, de la misma forma que en la clase anterior, recibe el ancho y el alto del laberinto donde se jugará. Establece el laberinto aleatoriamente, y coloca las posiciones de entrada y salida de la siguiente forma:

- La entrada al laberinto (punto de inicio del jugador) está en la primera casilla libre de la fila superior, empezando a recorrerla desde la izquierda. Si no hay ninguna casilla transitable, será la primera de la segunda fila, etc.
- La salida del laberinto está en la última casilla libre de la fila inferior, de tal forma que si la esquina inferior derecha del laberinto es pasillo, ésta será la salida del laberinto. Si ninguna de las casillas de la última fila es transitable, se buscará en la penúltima fila, y así sucesivamente.

El jugador empieza en la posición de entrada al laberinto.

Clase de test: `lps.pr1.logica.testsprofesor.PartidaTestAPI2`

3.3. Clase `AplicacionAutomatica`

En el paquete `lps.pr1.aplicacion`, se debe crear una nueva clase llamada `AplicacionAutomatica` que hereda de la `Aplicacion` ya implementada.

Las diferencias entre ambas son:

- El método `pideDireccion`, en vez de preguntar al usuario, utiliza la técnica de intentar siempre ir a la izquierda para recorrer el laberinto.

[4]El algoritmo debe funcionar incluso cuando el número aleatorio no lo es tal, por ejemplo, si la función `eligeEntero` devuelve siempre 0.

De esta forma, si por ejemplo se llegó a la casilla actual desde el Sur, intentará ir al Oeste; si no puede, intentará ir al Norte, y si no puede, al Este. Por último, si ninguna de las tres opciones es válida, volverá a ir hacia el Sur.

Al empezar la partida, siempre intenta primero ir *hacia el Oeste*.

- El método `iniciarPartida` en vez de pedir el nombre de un fichero, inicializa una partida aleatoria, con un tamaño que se pregunta al usuario.

Clase de test: `lps.pr1.aplicacion.testsprofesor.AplicacionAutomaticaTest`

4. Ejecutando los tests

En todas las clases anteriores se ha indicado una clase de test que prueba (al menos parcialmente) la funcionalidad de la clase.

Todas las clases de tests de esta práctica se proporcionan empaquetadas en `testPr1.jar`.

Para la ejecución de los tests, debe estar en el CLASSPATH tanto el código de la práctica compilado, como el JUnit.

Si estamos en la consola en el directorio del proyecto, que contiene la carpeta `./bin` con el código de la práctica compilada, podremos por ejemplo:

- Ejecutar *todos* los tests:

```
java -cp "junit-4.4.jar;./bin;testsPr1.jar"
      lps.pr1.testsprofesor.AllTests
```

- Ejecutar el test de la clase `Dirección`:

```
java -cp "junit-4.4.jar;./bin;testsPr1.jar"
      lps.pr1.logica.testsprofesor.DireccionTestAPI
```

Práctica 2: Conecta 4

Fecha de entrega: 24 de Enero de 2008

Material proporcionado:

Fichero	Explicación
`testPr2.jar`	Tests de algunas clases de la práctica que deben ejecutarse satisfactoriamente.
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega. También puede utilizarse su objetivo `runTestsProfesor` para ejecutar todos los tests anteriores sobre la práctica.
`jargs.jar`	Biblioteca para análisis de los parámetros en la línea de comandos de la aplicación.

1. Descripción

Se trata de implementar una aplicación que permita jugar a dos personas al *conecta 4*.

El conecta 4 es un juego de tablero en el que los jugadores, en turnos, van dejando caer sus fichas en un tablero vertical de siete columnas y seis filas. El objetivo del juego es conectar cuatro fichas de tu color, ya sea en vertical, horizontal o diagonal. Normalmente se juega con fichas rojas y amarillas y en nuestra aplicación siempre empezarán las *amarillas*.

La aplicación debe permitir jugar tanto en un interfaz de consola como en ventana gráfica.

Para el caso del interfaz de consola, el funcionamiento será similar al de la práctica 1: se irá preguntando a los jugadores en qué columna colocan la

ficha, comprobando su validez, y dibujando acto seguido el tablero. Cuando la partida termina, se indica el vencedor y se permite la opción de jugar de nuevo.

Por su parte, la ventana deberá tener *al menos*:

- Un menú con las opciones "Archivo" y "Ayuda". El primero de ellos permitirá empezar una partida nueva y terminar la actual; el segundo permitirá ir a la ventana "Acerca de", con la descripción del proyecto.
- El tablero de juego. Se podrá dibujar utilizando líneas y círculos (con los métodos de la clase `Graphics`) o utilizando imágenes externas que hacen de tablero y de fichas. Cuando el usuario pulse en una columna, se colocará la ficha en ella.
- Una etiqueta o barra de estado que muestre el estado del juego. Indicará en cada momento, si la partida aún no ha comenzado, de quién es el turno, o quién ha ganado.

El método de ejecución se configurará mediante parámetros a la aplicación, utilizando la opción `-i` o `--interface` de tal forma que si se ejecuta con[1]:

```
java lps.pr2.Main -i consola
```

se utilizará el interfaz de consola; mientras que utilizando:

```
java lps.pr2.Main -i swing
```

se utilizará el interfaz gráfico[2].

Para la interpretación de la línea de comandos se deberá utilizar la biblioteca *JArgs*, cuyo `.jar` y documentación está disponible en `http://jargs.sourceforge.net/`[3]

2. Diseño de clases

En este apartado aparecen las clases mínimas que la práctica debe implementar, junto con una breve explicación de ellas. El resto de clases se

[1] La orden exacta de ejecución puede variar para incluir opciones pasadas a `java`, como el CLASSPATH. Lo que no variará será la clase principal de la aplicación, que será `lps.pr2.Main`.

[2] También se podrá utilizar la versión larga, `java lps.pr2.Main --interface consola` o `swing`.

[3] Se puede configurar el proyecto de Eclipse para que encuentre la librería, accediendo a las propiedades del proyecto, y añadiendo el `.jar` al conjunto de librerías de la opción "Java Build Path".

deja a voluntad de los alumnos si bien se recomienda hacer un diseño de clases extensible (ver apartado 4 para más detalles). Todas las clases deberán aparecer dentro del paquete `lps.pr2`; también se permite el uso del paquete `lps.util` si se programan clases que se prevean útiles para otras prácticas.

2.1. Punto de entrada a la aplicación

Como ya ha quedado reflejado en la explicación anterior sobre los parámetros, el punto de entrada o clase principal de la práctica *debe* ser la clase `Main` del paquete `lps.pr2`.

El método `main` de esa clase deberá interpretar los argumentos según lo descrito anteriormente y lanzará el juego o bien en consola o bien en ventana. En caso de existir algún error en los parámetros, deberá mostrar una ayuda indicando el modo de uso correcto[4].

2.2. Aplicación

Para que la programación de la función `main` anterior sea sencilla, deberá existir el *interfaz* `lps.pr2.aplicacion.Aplicacion`, que tendrá los siguientes métodos:

- `init()`: inicializa la aplicación, creando todos los recursos necesarios para su correcta ejecución. Si hay algún error que impida su correcto funcionamiento (faltan recursos por ejemplo), debe devolver `false`.
- `run()`: lanza la ejecución de la aplicación.

En el mismo paquete, existirán dos implementaciones distintas del interfaz:

- `lps.pr2.aplicacion.AplicacionConsola`: que ejecuta el juego en modo consola.
- `lps.pr2.aplicacion.AplicacionSwing`: que ejecuta el juego en modo ventana, haciendo uso de swing.

De esta forma, el método `main` que lanza la aplicación, tendrá la siguiente estructura:

[4]Desde Eclipse se puede configurar la ejecución/depuración de la aplicación para que sea lanzada con parámetros. Para eso, en la ventana de gestión de las configuraciones de ejecución (Run >Run...), se pueden especificar los argumentos del programa en la pestaña "Arguments".

```
public static void main(String args[]) {
  // Análisis de los parámetros.
  // ...

  // Lanzamiento de la aplicación
  lps.pr2.aplicacion.Aplicacion app;

  if (!interfazGrafico)
    app = new lps.pr2.aplicacion.AplicacionConsola();
  else
    app = new lps.pr2.aplicacion.AplicacionSwing();

  if (!app.init()) {
    // Manejo del error.
    ...
  }
  app.run();
}
```

2.3. Lógica

Dentro del paquete `lps.pr2.logica` estarán las clases relacionadas con la lógica del juego.

Las clases mínimas que deben existir aparecen a continuación.

2.3.1. `Ficha`

Es un enumerado que contiene el tipo de fichas (rojas y amarillas). Para facilitar almacenar el tablero, el enumerado también tendrá el símbolo de ficha "vacía".

Los símbolos del enumerado *deberán* ser `VACIA`, `NEGRA` y `BLANCA`. Observa que el nombre del símbolo no coincide con el color real utilizado en la interfaz gráfica.

2.3.2. `Tablero`

Esta clase es la responsable de almacenar la configuración del tablero en el que se juega. Se recuerda que en el Conecta 4 el tablero tiene 6 filas y 7 columnas. Para acceder a ellas, se considerará que la celda de la esquina superior izquierda es la (0, 0).

Deberá tener, al menos, los siguientes métodos:

- Un constructor sin parámetros que inicializa el tablero sin fichas.
- `getCasilla` que devuelve la `Ficha` que está en la columna y fila especificada en los parámetros (la columna será el primer elemento).

- `ponerFicha`: recibe el color de una `Ficha` y la columna en la que se quiere poner la ficha, y la añade al tablero teniendo en cuenta las reglas concretas del Conecta 4. Si la ficha no puede colocarse (no entra en la columna, o ésta es incorrecta), devuelve `false`.
- `cuatroEnRaya`: analiza el tablero y devuelve `true` si el jugador con el color indicado mediante un parámetro ha conseguido hacer cuatro en raya, ya sea en horizontal, en vertical o en diagonal.

2.3.3. Partida

Representa una partida del Conecta 4. Tiene, al menos, los siguientes métodos:

- Un constructor sin parámetros. Para empezar una partida, habrá que invocar al siguiente método.
- `iniciarPartida`: inicializa los atributos para que comience una nueva partida.
- `getTurno`: devuelve el color (`Ficha`) del jugador que tiene el turno.
- `ponFicha`: recibe como parámetro la columna en la que se desea poner, y devuelve en un `booleano` el éxito o fracaso de la operación.
- `terminado`: devuelve `true` si la partida ha terminado.
- `ganador`: si `terminado()==true`, devuelve el color del jugador que ha ganado (o `Ficha.VACIA` si ha terminado en tablas). Si la partida no ha terminado devuelve `Ficha.VACIA`.
- `getTablero`: devuelve el tablero donde se está jugando.

La clase tendrá además, dos métodos `addObserver` y `removeObserver` explicados en la sección siguiente.

2.3.4. `ObservadorPartida`

Éste es un interfaz que permite a cualquier clase que lo implemente enterarse de los eventos que tienen lugar en la partida. En particular, el interfaz tiene los siguientes métodos, que son invocados desde la partida cuando suceden los eventos correspondientes:

- `void partidaEmpezada()`: indica que la partida acaba de comenzar.
- `void partidaTerminada()`: indica que la partida ha terminado, bien sea por tablas (no se pueden poner más fichas), bien porque hay un ganador.

- `void movimientoRealizado(Ficha, int)`: indica que se ha colocado una ficha en una columna determinada del tablero.

Para permitir a quien esté interesado `registrarse` en los eventos de la partida, la clase `Partida` debe disponer de los métodos antes mencionados:

- `void addObserver(ObservadorPartida)`: añade un nuevo observador de la partida, para ser informado de todos los eventos que ocurran. No se permiten registros múltiples, es decir, si un observador se registra dos o más veces, sólo será avisado una vez.
- `void removeObsever(ObservadorPartida)`: elimina un observador.

3. Tests de unidad

Junto con el enunciado se proporciona una biblioteca de clases con los tests de unidad que *debe* pasar la práctica. Los tests se ejecutan con el objetivo `runTestsProfesor` del script de Ant `build.xml` dado.

Se recuerda que los test anteriores *no* son infalibles, por lo que el funcionamiento de la práctica no está garantizado al pasarlos.

Los alumnos deberán implementar sus propios test de unidad en el directorio `./tests`, situado en el mismo directorio que el de código fuente, `./src`. Se debe programar la clase `lps.pr2.tests.AllTests`, que disponga del método

```
public static Test suite();
```

que devuelva la batería de pruebas implementadas para la práctica.

Para ejecutar los test implementados, se puede utilizar el objetivo `runTests` del script de Ant.

4. Recomendaciones

Es aconsejable de cara a las prácticas siguientes y al examen realizar un diseño de clases extensible, código claro y bien documentado. En particular, se aconseja:

- Utilizar el patrón Model-View-Controller en la implementación del entorno gráfico.
- Documentar el código siguiendo el formato de javadoc.
- Implementar tests de unidad de las clases que se programen, especialmente aquellas no relacionadas con swing.

Práctica 3: Complica

Fecha de entrega: 6 de Marzo de 2008

Material proporcionado:

Fichero	Explicación
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega.

1. Descripción

Complica es un juego diseñado por Reiner Knizia que puede verse como una variante del Conecta 4. Las diferencias son:

1. El tablero tiene 7 filas y 4 columnas.
2. Un jugador puede poner una ficha *en una columna completa*. En ese caso, la última ficha de la columna (la que está en la base) *sale* del tablero, desplazando a todas las que tiene encima.

La segunda de las diferencias provoca que, en ocasiones, *ambos* jugadores consigan cuatro fichas de su color seguidas. En ese caso, *la partida continúa* hasta que solamente uno de ellos tiene cuatro en raya.

Se trata de *extender* la implementación del *Conecta 4* de la práctica anterior para que también se permita jugar a *Complica*. Tanto el manejo mediante consola como el de ventana deben mantenerse, y su funcionamiento debe ser similar al de la práctica anterior.

El método de ejecución se configurará mediante parámetros a la aplicación:

- Utilizando la opción `-i` o `--interface` se selecciona la consola o swing.
- Utilizando la opción `-j` o `--juego` se selecciona el tipo de juego (`conecta4` o `complica`).

Por ejemplo, la orden[1]:

```
java lps.pr3.Main -i consola --juego conecta4
```

permite jugar al *Conecta 4* en consola, mientras que la orden

```
java lps.pr3.Main --interface swing -j complica
```

lanza la aplicación en modo gráfico para jugar a *Complica*.

Para la interpretación de la línea de comandos se deberá utilizar la biblioteca *JArgs*, cuyo `.jar` y documentación está disponible en `http://jargs.sourceforge.net/`.

2. Implementación

El diseño de clases se deja a voluntad de los alumnos si bien se recomienda hacer un diseño de clases extensible. Todas las clases deberán aparecer dentro del paquete `lps.pr3`; también se permite el uso del paquete `lps.util` si se programan clases que se prevean útiles para otras prácticas.

Aunque en el apartado 4 se indican una serie de recomendaciones de diseño, la principal es implementar la práctica utilizando el patrón MVC, de tal forma que ni la vista ni el controlador utilizado cambien con el tipo de juego.

Como ya ha quedado reflejado en la explicación anterior sobre los parámetros, el punto de entrada o clase principal de la práctica *debe* ser la clase `Main` del paquete `lps.pr3`.

El método `main` de esa clase deberá interpretar los argumentos según lo descrito anteriormente y lanzará el juego o bien en consola o bien en ventana. En caso de existir algún error en los parámetros, deberá mostrar una ayuda indicando el modo de uso correcto.

3. Tests de unidad

Los alumnos deberán implementar sus propios test de unidad en el directorio `./tests`, situado en el mismo directorio que el de código fuente,

[1] La orden exacta de ejecución puede variar para incluir opciones pasadas a `java`, como el CLASSPATH. Lo que no variará será la clase principal de la aplicación, que será `lps.pr2.Main`.

`./src`. Se debe programar la clase `lps.pr3.tests.AllTests`, que disponga del método

```
public static Test suite();
```

que devuelva la batería de pruebas implementadas para la práctica. Esta batería de pruebas *debe* incluir los tests implementados en la práctica anterior, modificándolos de acuerdo a los posibles cambios que hayan podido sufrir las clases y métodos implicados.

Para ejecutar los test implementados, se puede utilizar el objetivo `runTests` del script de Ant.

4. Recomendaciones

Es aconsejable de cara a las prácticas siguientes y al examen realizar un diseño de clases extensible, código claro y bien documentado. Se recuerda que el diseño de clases implementado *se evalúa* en el exámen.

Para poder reaprovechar las clases implementadas en la práctica anterior, se recomienda hacer uso de la *refactorización* que permite Eclipse. Las opciones de refactorización aparecen en la barra de menú, bajo la opción "Refactor", o en el menú contextual.

Consejos:

- Ampliar la clase `Tablero` para que permita tableros de distintos tamaños.
- Utilizar el patrón *estrategia* para definir el modo en el que se añaden las fichas en el tablero. Para ello, se puede crear un interfaz que defina la estrategia de colocación, `MetodoColocacionFicha` con un método:

  ```
  /**
   * Coloca la ficha siguiendo la estrategia concreta.
   * Si no se puede colocar, genera una excepción.
   * @param f Color de la ficha que se coloca.
   * @param col Columna donde se coloca.
   */
  void colocaFicha(Tablero t, Ficha f, int col)
                                  throws MovimientoNoValido;
  ```

  El método es invocado desde `ponerFicha` del `Tablero`.
- Implementar el método `ponFicha` utilizando el patrón *Template Method*, de tal forma que:
  - La clase `Partida` pasa a ser *abstracta*, a falta de la implementación de los métodos invocados desde el *template method*.

    - Se creen dos clases derivadas, `PartidaConecta4` y `PartidaComplica`, que implementan los métodos.

- Cambiar la vista del tablero, para que pueda pintar tableros de tamaños arbitrarios.
- Añadir un enumerado `TipoJuego` en el paquete `aplicacion`, que defina dos símbolos, uno para cada juego.
- Hacer que el método `init` de `Aplicacion` reciba el tipo de juego al que hay que jugar, de acuerdo a los parámetros pasados en la línea de comandos.

5. Instrucciones de entrega

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

Sólo uno de los dos miembros del grupo debe hacerlo, subiendo al campus un fichero llamado `grupoNN.zip`, donde `NN` representa el número de grupo con dos dígitos.

El fichero debe tener al menos el siguiente contenido[2]:

- Directorio `src` con el código de todas las clases de la práctica.
- Directorio `tests` con el código de los tests implementados por los alumnos.
- Directorio `bin` con el código compilado, tanto el de la aplicación como el de los tests.
- Directorio `doc` con la documentación generada utilizando `javadoc`.
- Fichero `pr3.jar` que empaqueta todas las clases de la práctica.

Recuerda que puedes utilizar Ant para generar los ficheros anteriores, así como para probar los tests proporcionados.

[2]Puedes incluir también ficheros de Eclipse como el directorio `.metadata`, etc.

Práctica 4: Gravity

Fecha de entrega: 10 de Abril

Material proporcionado:

Fichero	Explicación
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega.

1. Descripción

Los dos juegos de tablero *Conecta 4* y *Complica* tienen el concepto de *gravedad* que provoca que las fichas, al colocarse en la casilla "superior", descienda hasta la casilla libre más baja de esa columna.

La idea de gravedad puede generalizarse para crear juegos cuyas fichas, una vez colocadas en una casilla, "caen" en distintas direcciones.

Uno de esos juegos es *Gravity*. El juego recuerda mucho al *Conecta 4*, pero extendiendo la gravedad a otras direcciones.

Las diferencias principales de *Gravity* con el *Conecta 4* son las siguientes[1]:

- El tablero tiene 10 filas y 10 columnas.
- Un jugador puede poner ficha en *cualquier* casilla libre del tablero.
- Una vez colocada la ficha, ésta es "atraída" por el lado más cercano, en vez de por el lado inferior (como lo hace en el *Conecta 4*). Si la ficha es equidistante a dos de los lados, la gravedad le atrae hacia ambos,

[1] La regla para la victoria sigue siendo la misma: gana el primer jugador en hacer cuatro en raya.

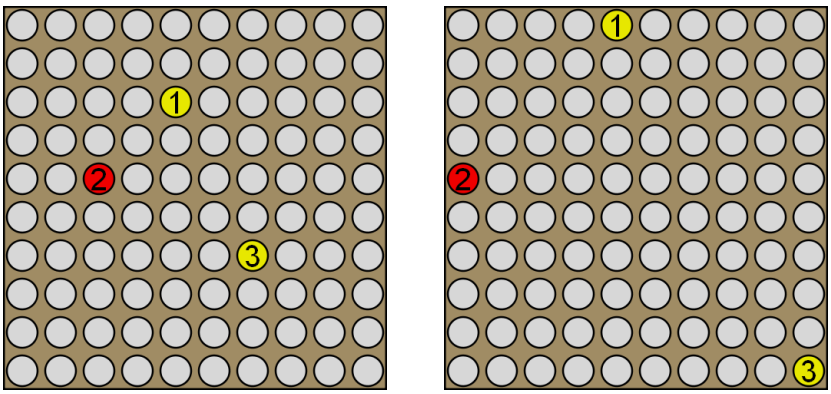

Figura 1: Ejemplo de movimiento en *Gravity*

debido a que los "vectores de fuerza" se suman, guiando a la ficha por la diagonal. Si la ficha es equidistante a tres lados, dos de los vectores de gravedad se anularán al tener sentidos opuestos y la ficha seguirá la tercera de las direcciones.

En la figura 1 pueden verse tres ejemplos de movimiento (el tablero izquierdo representa las fichas en las posiciones donde las coloca el jugador, y el tablero derecho las posiciones donde finalmente van a parar). La ficha 1 aparece colocada más cerca del borde superior, por lo que es atraída hacia él, mientras que la ficha 2 es atraída por el borde izquierdo. Por su parte, la ficha 3 es equidistante al borde derecho y al inferior, por lo que la dirección que toma en su "caída" es diagonal.

De manera intuitiva, puede verse el tablero como una pirámide de base cuadrangular. Al colocar una ficha, ésta se ve afectada por la inclinación del lado de la pirámide, y cae hasta que encuentra otra ficha o el lado.

Se trata de *extender* la implementación de la práctica anterior para que también permita jugar a *Gravity* con un tamaño de tablero *arbitrario* (y *no* únicamente con un tablero de 10x10).

El método de ejecución se configurará mediante parámetros a la aplicación:

- Utilizando la opción `-i` o `--interface` se selecciona la consola o swing. Observa que en el modo consola, se deberá preguntar al usuario la *posición completa* donde quiere colocar la ficha (y no únicamente la columna como ocurría cuando únicamente se jugaba a *Conecta 4* o *Complica*).
- Utilizando la opción `-j` o `--juego` se selecciona el tipo de juego (`conecta4`, `complica` o `gravity`).

- En caso de seleccionar este último juego, se admite especificar el tamaño del tablero con los parámetros `-f` o `--filas` y `-c` o `--columnas`. En caso de no hacerlo, se asumirá un tablero de 10x10.

  El tamaño del tablero debe ser como mínimo de 5x5 y como máximo 15x15. No se exige, no obstante, que el tablero sea cuadrado.

Por ejemplo, la orden[2]:

```
java lps.pr4.Main -i consola --juego conecta4
```

permite jugar al *Conecta 4* en consola, mientras que la orden

```
java lps.pr4.Main --interface swing -j complica
```

Por último

```
java lps.pr4.Main --interface swing --juego gravity -c 15
```

lanza la aplicación para jugar a *Gravity* en un tablero de 15 columnas y 10 filas.

Para la interpretación de la línea de comandos se deberá utilizar la biblioteca *JArgs*, cuyo `.jar` y documentación está disponible en `http://jargs.sourceforge.net/`.

2. Implementación

El diseño de clases se deja a voluntad de los alumnos si bien se recomienda hacer un diseño de clases extensible (ver apartado 4 para más detalles). Todas las clases deberán aparecer dentro del paquete `lps.pr4`; también se permite el uso del paquete `lps.util` si se programan clases que se prevean útiles para otras prácticas.

Como ya ha quedado reflejado en la explicación anterior sobre los parámetros, el punto de entrada o clase principal de la práctica *debe* ser la clase `Main` del paquete `lps.pr4`.

El método `main` de esa clase deberá interpretar los argumentos según lo descrito anteriormente y lanzará el juego o bien en consola o bien en ventana. En caso de existir algún error en los parámetros, deberá mostrar una ayuda indicando el modo de uso correcto.

Conviene hacer notar que el usuario puede especificar tamaños de tablero que tengan *casilla central* (como un tablero de 11x11). En ese caso, la ficha colocada en el centro es equidistante a *todos* los lados, por lo que su posición final no variará, ya que es atraída con la misma fuerza por todos los lados.

[2] La orden exacta de ejecución puede variar para incluir opciones pasadas a `java`, como el CLASSPATH. Lo que no variará será la clase principal de la aplicación, que será `lps.pr2.Main`.

3. Tests de unidad

Los alumnos deberán implementar sus propios test de unidad en el directorio `./tests`, situado en el mismo directorio que el de código fuente, `./src`. Se debe programar la clase `lps.pr4.tests.AllTests`, que disponga del método

```
public static Test suite();
```

que devuelva la batería de pruebas implementadas para la práctica. Esta batería de pruebas *debe* incluir los tests implementados en la práctica anterior, modificándolos de acuerdo a los posibles cambios que hayan podido sufrir las clases y métodos implicados (cambios de nombre de clases o

Para ejecutar los test implementados, se puede utilizar el objetivo `runTests` del script de Ant.

4. Recomendaciones

Es aconsejable de cara a las prácticas siguientes y al examen realizar un diseño de clases extensible, código claro y bien documentado. Se recuerda que el diseño de clases implementado *se evalúa* en el exámen.

Para poder reaprovechar las clases implementadas en la práctica anterior, se recomienda hacer uso de la *refactorización* que permite Eclipse. Las opciones de refactorización aparecen en la barra de menú, bajo la opción "Refactor", o en el menú contextual.

Consejos:

- Crear en el paquete `logica` una clase `Posicion` que almacene una posición dentro del tablero.
- Cambiar el método `ponerFicha` del tablero para que reciba la posición, en vez de únicamente la columna.
- Cambiar el método `ponerFicha` de `Tablero` para que devuelva la *posición final* en la que ha quedado la ficha colocada. Si la ficha no ha podido colocarse el método generará una excepción.
- Cambiar el observador de la partida para que los observadores:
  - Reciban la posición final de la ficha.
  - Reciban quién ha ganado cuando la partida ha concluido.

  De esta forma, ninguna vista necesita guardar una referencia al modelo, ya que pueden replicar su estado en los atributos de la propia vista.

Para que la vista del tablero funcione para los tres tipos de juego, y en especial, para el *Complica*, cuando recibe la notificación de que se ha puesto en una casilla ya ocupada, desplaza toda la columna hacia abajo para hacer hueco[3].

- Una nueva clase `PartidaGravity` es la responsable de las partidas del nuevo juego.
- Habrá que añadir un símbolo nuevo al enumerado `TipoJuego`, y extender los `init` de las `Aplicacione`s de acuerdo a él.

5. Instrucciones de entrega

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

Sólo uno de los dos miembros del grupo debe hacerlo, subiendo al campus un fichero llamado `grupoNN.zip`, donde `NN` representa el número de grupo con dos dígitos.

El fichero debe tener al menos el siguiente contenido[4]:

- Directorio `src` con el código de todas las clases de la práctica.
- Directorio `tests` con el código de los tests implementados por los alumnos.
- Directorio `bin` con el código compilado, tanto el de la aplicación como el de los tests.
- Directorio `doc` con la documentación generada utilizando `javadoc`.
- Fichero `pr4.jar` que empaqueta todas las clases de la práctica.

Recuerda que puedes utilizar Ant para generar los ficheros anteriores, así como para probar los tests.

[3] Observese que esto es compatible con el resto de juegos, que nunca notificarán una ficha puesta en una casilla ya ocupada.

[4] Puedes incluir también ficheros de Eclipse como el directorio `.metadata`, etc.

Práctica 5: Jugadores automáticos

Fecha de entrega: 30 de Abril

Material proporcionado:

Fichero	Explicación
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega.
`apache-ant-1.7.0-bin.zip`	Distribución de Ant 1.7.0.
`antvars.bat`	Fichero para configurar una consola de los laboratorios para poder ejecutar Ant, una vez descomprimido en `hlocal`.

1. Descripción

Esta práctica consiste en *extender* la implementación de la práctica anterior para permitir jugar a cualquiera de los tres juegos de tablero *Conecta 4*, *Complica* o *Gravity* teniendo como adversario la máquina.

También se podrá seguir jugando de la misma forma que se ha hecho hasta ahora, así como poner a dos jugadores controlados por el ordenador a jugar.

El método de ejecución se configurará mediante parámetros a la aplicación (los parámetros pueden utilizarse *en cualquier orden*):

- Utilizando la opción `-i` o `--interface` se selecciona la consola o swing.

- Utilizando la opción `-j` o `--juego` se selecciona el tipo de juego (`conecta4`, `complica` o `gravity`).
- En caso de seleccionar este último juego, se admite especificar el tamaño del tablero con los parámetros `-f` o `--filas` y `-c` o `--columnas`. En caso de no hacerlo, se asumirá un tablero de 10x10.

  El tamaño del tablero debe ser como mínimo de 5x5 y como máximo 15x15. No se exige, no obstante, que el tablero sea cuadrado.
- Utilizando la opción `-r` o `--rojas` se puede indicar qué tipo de jugador utilizará fichas rojas; de la misma forma, utilizando la opción `-a` o `--amarillas` se podrá indicar qué tipo de jugador utilizará las fichas amarillas. Los valores de ambas opciones podrán ser `humano` cuando se quiera indicar que el jugador es controlado por el usuario, e `ia` cuando se desee que el jugador esté controlado por la inteligencia artificial programada. En caso de no indicarse el jugador asociado a un color, se entenderá que es controlado por el usuario.

Si los parámetros son erróneos, la aplicación *debe* mostrar un texto de ayuda explicando el uso de los parámetros y acabar.

Como ejemplo de uso, la orden[1]

```
java lps.pr5.Main -i consola --juego conecta4
```

permite jugar al *Conecta 4* en consola a dos usuarios, mientras que la orden

```
java lps.pr5.Main -j complica --rojas ia --interface swing
```

se utiliza para jugar al *Complica* en un interfaz de ventana contra las rojas manejadas por el ordenador.

Por último

```
java lps.pr5.Main -r ia --interface swing --juego gravity -c 15
                  --amarillas ia
```

lanza la aplicación para jugar a *Gravity* en un tablero de 15 columnas y 10 filas de tal forma que ambos jugadores son controlados por la máquina.

Para la interpretación de la línea de comandos se deberá utilizar la biblioteca *JArgs*, cuyo `.jar` y documentación está disponible en `http://jargs.sourceforge.net/`.

[1] La orden exacta de ejecución puede variar para incluir opciones pasadas a `java`, como el CLASSPATH. Lo que no variará será la clase principal de la aplicación, que será `lps.pr5.Main`.

Una vez terminada una partida, la aplicación debe permitir jugar otra, bien preguntando directamente o bien mediante una opción del tipo "Iniciar partida" en el menú.

Es importante hacer notar que en el caso de utilizar *swing*, cuando el jugador controlado por la máquina está decidiendo qué movimiento realizar, la ventana de Swing *debe* seguir respondiendo a los eventos del usuario (mover la ventana, minimizarla, etc.). Se permite, no obstante, que la opción "Terminar partida" del menú no actúe hasta que el jugador que tiene el turno no mueva.

2. Implementación

El diseño de clases se deja a voluntad de los alumnos si bien se recomienda hacer un diseño de clases extensible (ver apartado 4 para más detalles). Todas las clases deberán aparecer dentro del paquete `lps.pr5`; también se permite el uso del paquete `lps.util` si se programan clases que se prevean útiles para otras prácticas.

Para la implementación de los jugadores controlados por la máquina se debe hacer uso del algoritmo *minimax* (con o sin poda *alfa-beta*).

Como ya ha quedado reflejado en la explicación anterior sobre los parámetros, el punto de entrada o clase principal de la práctica *debe* ser la clase `Main` del paquete `lps.pr5`.

El método `main` de esa clase deberá interpretar los argumentos según lo descrito anteriormente y lanzará el juego o bien en consola o bien en ventana. En caso de existir algún error en los parámetros, deberá mostrar una ayuda indicando el modo de uso correcto.

La práctica entregada *debe* compilar correctamente utilizando el script de Ant proporcionado[2]. Una vez compilado, los tests también deben poder ser ejecutados automáticamente utilizando el objetivo `runTests`.

3. Tests de unidad

Los alumnos deberán implementar sus propios test de unidad en el directorio `./tests`, situado en el mismo directorio que el de código fuente, `./src`. Se debe programar la clase `lps.pr5.tests.AllTests`, que disponga del método

```
public static Test suite();
```

que devuelva la batería de pruebas implementadas para la práctica.

[2]El script de Ant es compatible con la versión 1.7.

Para ejecutar los test implementados, se puede utilizar el objetivo `runTests` del script de Ant.

4. Recomendaciones

Es aconsejable de cara a las prácticas siguientes y al examen realizar un diseño de clases extensible, código claro y bien documentado. Se recuerda que el diseño de clases implementado *se evalúa* en el exámen.

Para la implementación de esta práctica es aconsejable cambiar el *modo de ejecución* de la aplicación con respecto a la práctica anterior.

Los siguientes apartados describen brevemente el diseño recomendado.

4.1. Control de ejecución

En la práctica anterior, el *control de ejecución* recaía o bien en la aplicación de consola (método `run`) o bien en el MVC gobernado por la hebra de Swing. Para esta práctica se recomienda que sea *la propia partida* la responsable de hacer que los jugadores vayan colocando sus fichas.

De esta forma, se elimina la necesidad de dos subclases distintas de `Aplicacion`, pues eran quienes distinguían entre las dos alternativas mencionadas.

4.2. Lógica

Para implementar lo anterior se aconseja lo siguiente:

- Crear en el paquete `logica` un interfaz `Jugador` que disponga de un método `dameMovimiento` que, dado el estado de un tablero, devuelva en qué posición pone el jugador.
- Crear una o varias clases que implementen el interfaz anterior y que codifiquen las distintas inteligencias para cada uno de los juegos posibles de la aplicación.
- Crear dos clases distintas para los dos tipos de jugadores humanos posibles: el que utiliza la consola para pedir la posición, y el que utiliza Swing. Hay que hacer notar que la implementación de sus métodos `dameMovimiento` (llamados desde la `Partida`) requerirán la intervención del usuario.
- Añadir un método `run` a la partida que vaya llamando alternativamente al método `dameMovimiento` del jugador que tiene el turno. Cuando la partida termine, preguntará si se desea jugar otra partida (ver sección 4.3 para más detalles sobre esto último).

Con estos cambios, algunos de los métodos que estaban disponibles en las prácticas anteriores en la clase `Partida` pasan a no ser necesarios, como por ejemplo `ponFicha` o `terminado`[3]. En particular, únicamente son necesarios los siguientes:

- `addObserver` y `removeObserver`: para gestionar los observadores de la partida.
- `getNumFilasTablero` y `getNumColumnasTablero`: dos métodos que devuelven el número de filas y columnas que tiene el tablero con el que se juega la partida.
- `setJugadores`: debe ser llamada antes del método `run`, y recibe los `Jugador`es de la partida.
- `run`: que va jugando partidas con los jugadores establecidos con el método anterior hasta que no se deseen jugar más, o se cancele la partida en curso.
- `solicitarTerminacion`: este método puede ser llamado desde otra hebra cuando la partida está ejecutando el método `run`. El método `run` terminará la partida en curso en cuanto le sea posible.
- `setPreguntarJugarOtra`: establece el modo en el que la partida preguntará si se desea jugar otra partida. La siguiente sección describe más detalles al respecto.

4.3. Interfaz de usuario

Con el modelo de ejecución anterior, es la propia `Partida` la que invoca al GUI ya sea a través de las invocaciones a sus observadores, cuando pregunta a los jugadores humanos por el siguiente movimiento o cuando pregunta si se desea jugar otra partida. Además, cuando se está utilizando *Swing*, el usuario también puede *cancelar* una partida en curso.

Sin embargo, la `Partida` *debe* seguir siendo independiente del interfaz de usuario utilizado. Esa independencia se consigue:

- Utilizando los observadores ya utilizados en prácticas anteriores.
- Utilizando los propios `Jugador`es humanos que contienen el código dependiente del interfaz para preguntar el siguiente movimiento.
- Utilizando el método `solicitarTerminacion` de la partida, que será invocado por la hebra de *Swing* directamente.

[3] No son necesarios en el sentido de que no es necesario que sean *públicos*.

- Utilizando un interfaz nuevo, `PreguntaJugarOtra` en el paquete `gui`. Este interfaz contiene un método `jugarOtra` que pregunta al usuario utilizando el interfaz seleccionado si se desea jugar otra partida o no. De esta forma el interfaz *abstrae* el GUI concreto usado. El método es invocado desde el método `run` de la `Partida` cuando se termina (ya sea porque hay ganador, tablas, o es cancelada por el usuario).

4.4. Factorías abstractas

Las distintas opciones que permite la aplicación hacen que el control de las distintas posibilidades sea ya demasiado complicado. En particular, existen dos ejes fundamentales de variabilidad: el tipo de juego seleccionado, y el interfaz de usuario utilizado.

Se recomienda utilizar el patrón *Abstract Factory* para cada una de ellas. De esta forma, se aconseja:

- Crear una factoría abstracta en el paquete `logica` que se instanciará en base al tipo de juego seleccionado y que tenga los siguientes métodos:
  - `creaPartida`: crea la partida que se jugará. Recibe el tamaño del tablero indicado en los argumentos de la aplicación.
  - `creaJugadorIA`: recibe el color de una ficha, y devuelve un `Jugador` que codifica la IA de la máquina para jugar al juego concreto.
  - `creaJugadorHumano`: crea el `Jugador` humano. Para poder hacerlo, necesitará recibir como parámetro la factoría abstracta que depende del interfaz de usuario y que se describe a continuación.
- Crear una factoría abstracta en el paquete `gui` que se instanciará en base al tipo de GUI seleccionado y que tenga los siguientes métodos:
  - `creaGUIPartida`: crea un objeto responsable de presentar el interfaz de usuario que presenta la partida. Para ello, será necesario crear un interfaz `GUIPartida` que se instancie para cada uno de los posibles interfaces y que tenga como métodos un `init` que reciba la partida (para que el GUI pueda hacerse observadora de ella), y un `getPreguntaJugarOtra`, que devuelva el objeto que pregunta al usuario si se quiere jugar otra partida o no.
  - `creaJugadorHumano`: devuelve un `Jugador` que, utilizando el GUI seleccionado, pregunta al usuario el siguiente movimiento. Puede recibir un parámetro `booleano` que indique si en el juego concreto la fila es significativa o no.

4.5. Aplicación

Con las clases anteriores, deja de ser necesaria la división entre la aplicación de consola y la aplicación swing de las prácticas anteriores, por lo que pueden suprimirse.

5. Instrucciones de entrega

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

Sólo uno de los dos miembros del grupo debe hacerlo, subiendo al campus un fichero llamado `grupoNN.zip`, donde `NN` representa el número de grupo con dos dígitos.

El fichero debe tener al menos el siguiente contenido[4]:

- Directorio `src` con el código de todas las clases de la práctica.
- Directorio `tests` con el código de los tests implementados por los alumnos.
- Directorio `bin` con el código compilado, tanto el de la aplicación como el de los tests.
- Directorio `doc` con la documentación generada utilizando `javadoc`.
- Fichero `pr5.jar` que empaqueta todas las clases de la práctica.

Recuerda que puedes utilizar Ant para generar los directorios `bin` y `doc`, así como para generar el fichero `pr5.jar` y para probar los tests. El contenido de los directorios `src` y `tests` debe permitir compilar, crear el `jar` y ejecutar los tests utilizando el script de Ant proporcionado. El fichero `jar` generado por Ant deberá permitir ejectuar la práctica con la orden

```
java -cp "pr5.jar;jargs.jar" lps.pr5.Main <parámetros>
```

para cualquier configuración de los parámetros.

[4] Puedes incluir también ficheros de Eclipse como el directorio `.metadata`, etc.

Práctica 6: Juego en red

Fecha de entrega: 29 de Mayo

Material proporcionado:

Fichero	Explicación
`build.xml`	Fichero de entrada a Ant que facilita la generación de ficheros necesarios para la entrega.

1. Descripción

Esta práctica consiste en *extender* la implementación de la práctica anterior para permitir jugar en red.

En particular, se deben mantener las capacidades de la práctica anterior. Además, se añadirá:

- Un nuevo tipo de jugador "remoto", que indica que el jugador *no* está controlado ni por la máquina ni por el usuario sentado enfrente de ella, sino por un usuario remoto conectado por red.
- Un nuevo tipo de partida "remota", que indica que la partida a la que se está jugando no se encuentra en la máquina local, sino en una máquina remota (servidora).

De esta forma, se distinguen dos tipos de ejecuciones[1]:

- La ejecución como máquina "servidora": contiene la implementación de la partida y, posiblemente, algún jugador configurado como "remoto".

[1] Existe una tercera aproximación mixta, como se ve en el Escenario 4 más adelante

- La ejecución como máquina "cliente": la partida está configurada como "remota".

Para indicar que un jugador es remoto, se indica con `remoto` en la descripción del jugador. Cuando la partida es remota, se indica utilizando como nombre del juego la cadena "`remoto:`" seguido del nombre del host. En ese caso, si no se describe qué tipo de jugadores se deben utilizar, se asumirá que ninguno de los jugadores es controlado por la máquina que lanza la aplicación.

Si la aplicación se utiliza como en las prácticas anteriores, donde los dos jugadores lo hacen en la propia máquina, se puede especificar el argumento `-n` o `--network` para permitir espectadores de la partida, es decir que otros usuarios desde sus máquinas puedan conectarse y ver la partida que están jugando. El número máximo de espectadores estará definido en una constante en el código a la que puede darse el valor 10.

Podemos ejemplificar la nueva funcionalidad de juego en red con cuatro escenarios distintos (recuerdese que todos los ejemplos indicados en el enunciado de las prácticas anteriores *deben* seguir funcionando).

Escenario 1: Un jugador humano jugando a *Complica* contra otro en otra máquina

El servidor se lanzaría con[2]:

```
java lps.pr6.Main -i swing -a humano -r remoto
```

y el cliente con

```
java lps.pr6.Main -j remoto:hostServidor -i swing -r humano
```

siendo `hostServidor` el nombre de la máquina que hace de servidor.

Escenario 2: dos IAs, una contra otra, en dos máquinas distintas, y el servidor en otra

Servidor:

```
java lps.pr6.Main -j gravity -i consola -a remoto -r remoto
```

Cliente 1 (se ve la partida en consola):

```
java lps.pr6.Main -i consola --amarillas ia
                  -j remoto:hostServidor
```

[2]La orden exacta de ejecución puede variar para incluir opciones pasadas a `java`, como el CLASSPATH. Lo que no variará será la clase principal de la aplicación, que será `lps.pr6.Main`, ni los parámetros pasados a ella.

Cliente 2 (se ve la partida en *swing*):

```
java lps.pr6.Main -r ia -i swing --juego remoto:hostServidor
```

Escenario 3: Dos jugadores en la misma máquina con un espectador externo

Servidor:

```
java lps.pr6.Main --network -j complica -r humano
                  --amarillas humano
```

Cliente (espectador):

```
java lps.pr6.Main -i swing --juego remoto:hostServidor
```

Escenario 4: Un cliente jugando con amarillas y haciendo de servidor para otro cliente que juega con rojas

Servidor:

```
java lps.pr6.main -j complica -r remoto -a remoto -i consola
```

Cliente 1 (hace de servidor para cliente 2)

```
java lps.pr6.main -i consola -j remoto:hostServidor
                  -a humano -r remoto
```

Cliente 2 (se conecta a cliente 1 en vez de al servidor)

```
java lps.pr6.main -j remoto:hostCliente1 -i swing -r ia
```

Como en las prácticas anteriores, si los parámetros son erróneos, la aplicación *debe* mostrar un texto de ayuda explicando el uso de los parámetros y acabar. Para la interpretación de la línea de comandos se deberá utilizar la biblioteca *JArgs*, cuyo `.jar` y documentación está disponible en `http://jargs.sourceforge.net/`.

Una vez terminada una partida, la aplicación debe permitir jugar otra. No obstante *únicamente* preguntará si se desea jugar otra partida en la máquina que hace las veces de *servidora* de la partida utilizando el interfaz elegido y sin importar que en ella haya algún jugador humano jugando. Las otras comenzarán automáticamente a jugar cuando ésta lo ordene.

2. Implementación

El diseño de clases se deja a voluntad de los alumnos si bien se recomienda hacer un diseño de clases extensible (ver apartado 4 para más detalles). Todas las clases deberán aparecer dentro del paquete `lps.pr6`; también se permite el uso del paquete `lps.util` si se programan clases que se prevean útiles para otras prácticas.

Los protocolos de comunicación, además, deben ser los descritos en la sección 5, de tal forma que puedan conectarse dos implementaciones distintas de la práctica y jugar entre ellas.

Como ya ha quedado reflejado en la explicación anterior sobre los parámetros, el punto de entrada o clase principal de la práctica *debe* ser la clase `Main` del paquete `lps.pr6`.

El método `main` de esa clase deberá interpretar los argumentos según lo descrito anteriormente y lanzará el juego o bien en consola o bien en ventana. En caso de existir algún error en los parámetros, deberá mostrar una ayuda indicando el modo de uso correcto.

La práctica entregada *debe* compilar correctamente utilizando el script de Ant proporcionado[3]. Una vez compilado, los tests también deben poder ser ejecutados automáticamente utilizando el objetivo `runTests`. El `.jar` generado con el script de Ant deberá ser directamente ejecutable, sin requerir ningún otro fichero adicional.

3. Tests de unidad

Los alumnos deberán implementar sus propios test de unidad en el directorio `./tests`, situado en el mismo directorio que el de código fuente, `./src`. Se debe programar la clase `lps.pr6.tests.AllTests`, que disponga del método

```
public static Test suite();
```

que devuelva la batería de pruebas implementadas para la práctica. Esta batería de pruebas *debe* incluir los tests implementados en la práctica anterior, modificándolos de acuerdo a los posibles cambios que hayan podido sufrir las clases y métodos implicados.

4. Recomendaciones

Es aconsejable realizar un diseño de clases extensible, código claro y bien documentado. Se recuerda que el diseño de clases implementado *se evalúa*

[3] El script de Ant es compatible con la versión 1.7.

en el exámen.

Para la implementación se aconseja crear un paquete `lps.pr6.net` que contenga todas las clases relacionadas con la gestión de la red.

En particular, se aconseja:

- Añadir a la factoría de la lógica un nuevo método `creaJugadorRemoto` para crear un jugador que no es controlado por un humano, ni por la IA, sino por un jugador situado en otra máquina distinta, es decir, que pregunte por red el siguiente movimiento.
- Crear una nueva clase que herede de la clase `Partida` y que represente una partida remota, es decir una partida que no está controlada en la máquina, sino en otra máquina servidora.
- Crear una nueva factoría lógica, `FactoriaPartidaRemota`, utilizada cuando se juega una partida cliente.
- Añadira a la factoría de lógica un método `creaJugadorPorDefecto`, que cree el tipo de jugador que debe utilizarse si no se indican parámetros en el ejecutable. En las factorías de los tres juegos se creará un jugador humano. En la factoría de la partida remota, no se creará ningún jugador, pues éste está por defecto controlado por el servidor.

5. Descripción de los protocolos de comunicación

5.1. Comunicación entre jugadores

Para la comunicación entre los jugadores se utiliza un puerto TCP para cada uno de ellos, de forma que la máquina servidora se quedará escuchando del puerto `8586` para la comunicación con el jugador *amarillo*, y el `8585` para la comunicación con el jugador rojo.

Cuando el jugador desde el lado del servidor recibe la orden de la partida de poner ficha (en `dameMovimiento`), éste enviará la solicitud por la red. La solicitud consistirá en una simple *serialización* del estado del tablero que recibe como parámetro. Esta serialización se realizará de la siguiente forma: todo el estado del tablero se codificará en una única línea formada por números separados por espacios. Los dos primeros contendrán el ancho y el alto del tablero. A continuación viene una descripción de cada una de las casillas, primero todas las de la primera fila, después los de la segunda y así sucesivamente. El número `0` indicará la casilla vacía, el `1` indicará una casilla amarilla, y por último el `2` indicará una casilla roja.

El cliente contestará con otra línea, indicando la posición donde se coloca la ficha, enviando primero la columna y después la fila, donde la columna estará comprendida entre `[0..numColumnas-1]` y la fila estará comprendida entre `[0..numFilas-1]`.

5.2. Comunicación entre partidas

Para la comunicación entre las partidas se utiliza el puerto TCP número `8587`. Cuando se lanza la aplicación que hace de servidor utilizando el parámetro `--network` o configurándola con algún jugador `remoto`, una hebra auxiliar se quedará escuchando del puerto para permitir a otras aplicaciones *conectarse* a la partida y ser informada de los eventos que suceden en ella.

Todos los eventos que ocurran en la partida que se está jugando deberán ser notificados a las aplicaciones conectadas, según el siguiente protocolo[4]:

- La cadena "`PE`" indica que la partida comienza.
- La cadena "`PT`" seguida de un color de ficha indica que la partida ha terminado. El color está codificado utilizando las cadenas "`vacia`" (sin acento), "`amarilla`" y "`roja`".
- La cadena "`MR`" se utiliza para notificar un movimiento realizado. En la misma línea aparecerá a continuación el color de la ficha que ha puesto ("amarilla" o "roja"), seguido de el número de columna (posición X) y de fila (posición Y)[5].

A su vez, *todos* los espectadores podrán solicitar la terminación de la partida, enviando la cadena "`TP`" al servidor.

Cuando un cliente se conecta al servidor, éste envía información sobre el tipo de partida que se está jugando. En particular, se recibe una línea cuyos dos primeros caracteres indican el tipo de juego, y que son seguidos por el tamaño del tablero en el que se está jugando separados por espacios. Para el *Conecta 4*, se enviará `C4`, para el *Complica* la codificación es `CO` y por último, en el *Gravity* se envía `GR`[6].

Dado que los espectadores pueden conectarse en cualquier momento de la partida, es decir, se han podido ya tener movimientos por parte de los jugadores, el servidor enviará acto seguido *todos* los eventos que han sucedido en la partida en curso, desde el `PE` que indica que la partida ha empezado.

6. Instrucciones de entrega

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

[4]Igual que en el caso anterior cada orden irá en una cadena terminada por un retorno de carro, \n.

[5]Igual que antes, la esquina superior izquieda será la posición (0, 0).

[6]Por lo tanto, para el caso del *Conecta 4* siempre se enviará la cadena `C4 7 6`, y para el caso del *Complica* siempre será `CO 4 7`.

Sólo uno de los dos miembros del grupo debe hacerlo, subiendo al campus un fichero llamado `grupoNN.zip`, donde `NN` representa el número de grupo con dos dígitos.

El fichero debe tener al menos el siguiente contenido[7]:

- Directorio `src` con el código de todas las clases de la práctica.
- Directorio `tests` con el código de los tests implementados por los alumnos.
- Directorio `bin` con el código compilado, tanto el de la aplicación como el de los tests.
- Directorio `doc` con la documentación generada utilizando `javadoc`.
- Fichero `pr6.jar` que empaqueta todas las clases de la práctica.

Recuerda que puedes utilizar Ant para generar los directorios `bin` y `doc`, así como para generar el fichero `pr6.jar` y para probar los tests. El contenido de los directorios `src` y `tests` debe permitir compilar, crear el `jar` y ejecutar los tests utilizando el script de Ant proporcionado. El fichero `jar` generado por Ant deberá permitir ejectuar la práctica con la orden

```
java -cp "pr6.jar;jargs.jar" lps.pr6.Main <parámetros>
```

para cualquier configuración de los parámetros.

[7] Puedes incluir también ficheros de Eclipse como el directorio `.metadata`, etc.