Grado en Ingeniería de la Salud

Metodología de la Programación

Práctica Obligatoria 2 – Segunda Convocatoria

Kamisado 4.0

Docentes:

Raúl Marticorena

Estrella Morales

Universidad de Burgos
Universidad de Burgos

Índice de contenidos

1. Introducción

El objetivo fundamental es continuar implementando una variante **simplificada** del juego **Kamisado, incluyendo dos tipos de partida: simple y estándar.**

Partiendo de lo realizado en la práctica anterior, se incluyen además los **siguientes conceptos:**

- Inclusión de una **jerarquía de herencia de torres y árbitros** (se recomienda revisar el Tema 4. Herencia de teoría).
- **Inclusión y uso** de **interfaces y clases genéricas** (se recomienda revisar el Tema 5. Genericidad de teoría).
- Inclusión de **excepciones** comprobables y su lanzamiento con un enfoque defensivo con el consiguiente tratamiento de excepciones: **comprobables y no comprobables** (se recomienda revisar el Tema 6. Programación Defensiva y Tratamiento de Excepciones de teoría).

A continuación se repasan las reglas del juego,ya tenidas en cuenta en la anterior versión (Kamisado 3.0), que son las aplicadas en un **partida simple**, para posteriormente explicar las diferencias en una **partida estándar.**

Partida simple

Kamisado es un juego abstracto de **tablero** de 8x8 celdas, para **dos jugadores**. Las **celdas** tienen un **color asignado fijo**, de entre solo **ocho colores** posibles (i.e. amarillo, azul, marrón, naranja, púrpura, rojo, rosa, y verde). Sobre dicho tablero se colocan **8 torres[1] blancas** en la fila superior del tablero, y **8 torres negras** en la fila inferior del tablero. A partir de ahora diremos que las **torres tienen un turno , blanco o negro,** según corresponda.

A cada jugador, se le asignan sus 8 torres correspondientes. Adicionalmente, cada **torre** además del turno (blanco o negro), tiene **asignado un color de entre esos ocho**. La colocación de las torres de cada turno, al inicio de la partida, coincide con el color de la celda. Por ejemplo, la torre negra de color amarillo, se colocará en la fila inferior, en la celda de color amarillo.

A continuación se muestra el aspecto que tendrá el juego iniciando la partida en **modo simple** (ver Ilustración 1):

1 Denominadas también “torres dragón”.

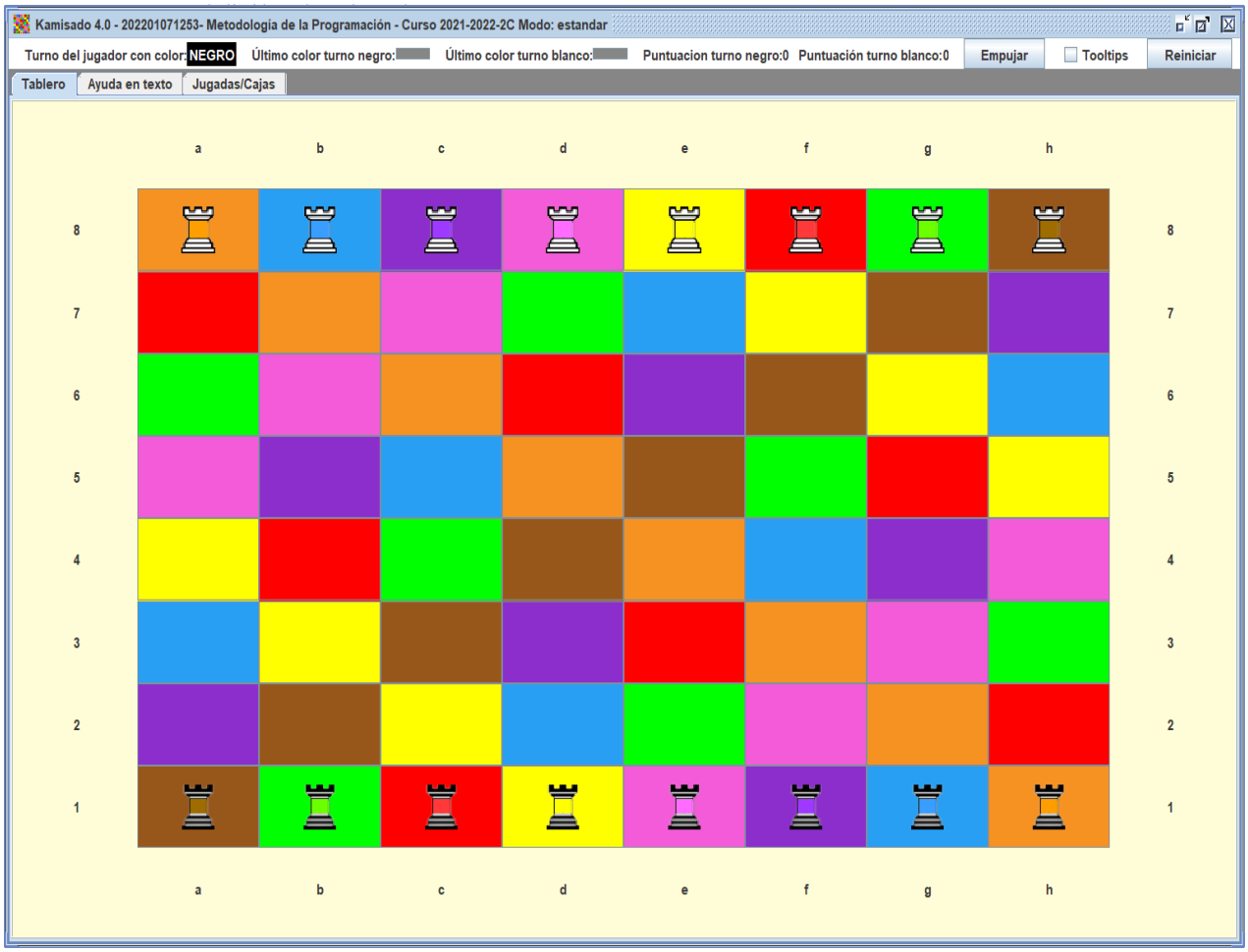


Ilustración 1: Interfaz gráfico de inicio de partida

Por simplificacion, siempre comienza la partida el jugador con turno negro. En este primer turno, puede mover discrecionalmente una de sus torres a otra celda vacía. El **color de la celda** donde se coloca la torre, **determina** que en el siguiente turno, el jugador **contrario** está obligado a mover su **torre de dicho color**.

Por lo tanto, solo es discrecional el primer movimiento de salida en cada ronda. El resto de movimientos están siempre **condicionados** por el **color de la celda** a donde **movió en último lugar el jugador contrario**.

Los **movimientos** de las torres están **limitados** por las siguientes reglas:

- Solo se puede mover una torre del turno actual hacia la fila de inicio del contrario, **solo en sentido vertical o diagonal** (i.e. solo se puede **avanzar** hacia la fila de partida del contrario, pero **nunca retroceder**).
- **No se puede saltar** sobre otras torres, independientemente del turno que tengan (ni siquiera sobre torres propias).
- **No se puede ocupar una celda que contenga otra torre**. En este juego no se "comen" o eliminan torres del contrario (ni propias).
- El jugador con turno está **obligado a mover** siempre que haya algún movimiento legal. **No puede pasar turno.**

La **partida simple finaliza** en la **primera ronda**, cuando uno de los jugadores consigue **colocar una de sus torres en la fila de salida del turno contrario sumando 1 punto.** A la partidas simple se las considera de ronda única, finalizando al conseguir un punto.

Para indicar las celdas del tablero, los jugadores utilizarán la misma notación utilizada en el ajedrez, denominada **"notación algebraica"**. Por ejemplo y según se muestra en la Ilustración 1, la celda [0][0] sería "a8", la celda [3][4] sería "e5" y la celda [6][5] sería "f2". Cuando se quiere mencionar una jugada completa, moviendo una torre de una celda origen a una celda destino, se indicarán las dos celdas seguidas. Por ejemplo: "a1c3" sería una jugada donde se mueve la torre desde la celda [7][0] a la celda en [5][2].

Si un jugador está obligado a mover una torre, y dicha torre está **bloqueada** (según las reglas), se considera que hace un movimiento de "distancia cero", colocando su torre en la **misma celda en la que estaba**, y por lo tanto el jugador contrario ahora tendrá que mover su torre del **color de dicha celda** en la que ha quedado bloqueado el contrario.

Si se diese la situación de que el **bloqueo se da en ambos jugadores**, denominado **bloqueo mutuo o *deadlock*,** se considera **finalizada**, dando como perdedor al jugador que provocó dicha situación con un movimiento de torre. Es decir, pierde el jugador que hizo el último movimiento que no fuera de "distancia cero".

Partida estándar

En esta segunda versión de la práctica, se amplía con la posiblidad de jugar **partidas estándar**, con **varias rondas.**

La partida se inicia igual que una partida simple. En una partida **estándar** las **torres** que alcanzan la fila del contrario **suman 1 punto** al turno correspondiente y además dicha torre se **transforma** en una "**torre sumo uno"** (en algunas versiones del juego se añade un "diente de dragón"), **finalizando la ronda** y **reiniciando las torres a la posición inicial para iniciar otra ronda**. Se colocan de nuevo las torres en la posición de partida, incluyendo ahora la nueva "torre sumo uno" en su color correspondiente. Inicia la **nueva ronda** el turno que perdió la ronda previa.

Si se alcanza la fila del contrario con una **torre sumo uno** se consiguen **3 puntos**.

Se considera que se **finaliza la partida** cuando un **turno consigue 3 o más puntos**, al sumar los puntos acumulados en distintas rondas con **torres simples** o **torres sumo uno**.

Las **torres sumo uno** tienen unas reglas de movimiento **adicionales**:

- Solo pueden desplazarse un máximo de una distancia de 5 celdas en cualquiera de los sentidos básicos (vertical o diagonal).
- Pueden "empujar" una posición hacia delante a torres del contrario que la bloqueen (denominado "empujón sumo"), pero solo en sentido vertical:
  - Solo pueden empujar una torre del turno contrario.
  - Detrás de esa torre empujada, debe haber una celda vacía. No se puede "empujar" o echar torres del turno contrario fuera del tablero.
  - No se puede empujar a otra "torre sumo uno" del contrario, solo a una torre simple.
  - Cuando se produce un "empujón sumo", el turno contrario pierde turno y vuelve a mover el turno que realizó el empujón.
  - El color de la torre a mover, tras el empujón, se obtiene del color de la celda donde ha quedado situada la torre del contrario.

Para desarrollar este juego se establece la siguiente estructura de paquetes y módulos/clases (ver Tabla 1):

Paquete	Módulos / Clases	Nº	Descripción

`juego.modelo`	`Celda`
	`Color`
	`Tablero`
	`Torre`
	`TorreAbstracta`
	`TorreSimple`
	`TorreSumoUno`
	`Turno`
		`1 interfaz`
		`1 clase abstracta`
		`4 clases concretas`
		`2 enumeraciones`
			Modelo fundamental.

`juego.control`	**`Arbitro`**
	`ArbitroAbstracto`
	`ArbitroEstandar`
	`ArbitroSimple`
		`1 interfaz`
		`1 clase abstracta`
		`2 clases concretas`
			Lógica de negocio (reglas del juego).

`juego.util`	`CoordenadasIncorrectasException` **`Sentido`**	`1 clase excepción` `1 enumeración`	Utilidad.
`juego.textui`	**`Kamisado`**	`1 clase`	Interfaz de usuario en modo texto. Se proporciona **parcialmente** el código a completar por el alumnado.
`juego.gui`	`Sin determinar`	`Sin determinar`	Interfaz gráfica (se proporcionan los binarios en formato `.jar`)
`juego.gui.images`	`Sin determinar`	`Sin determinar`	Imágenes en la aplicación (se proporcionan dentro del fichero `.jar`)

Tabla 1: Resumen de paquetes y módulos/clases del sistema

La interfaz gráfica se ha subcontratado a la empresa ECMA (filas en color verde en Tabla 1) y se proporciona el fichero binario en formato `.jar` en la plataforma UBUVirtual.

Es labor del alumnado implementar/completr los **ficheros fuente** **`.java`** restantes necesarios[2] para el correcto cierre y ensamblaje del sistema, tanto en modo texto y gráfico, junto con el resto de productos indicados en el apartado **3 Descripción.**

Para ello se deben seguir las indicaciones dadas y los diagramas disponibles, **respetando los diseños y signaturas de los métodos**, con sus correspondientes modificadores de acceso, quedando **a decisión del alumnado la inclusión de atributos y/o métodos privados o protected (`private` o `protected`), siempre de manera justificada. NO se pueden añadir atributos ni métodos amigables o públicos adicionales.**

2. Objetivos

- Construir siguiendo los diagramas e indicaciones dadas la implementación del juego en Java.
  - Completando una aplicación en modo texto.

2 Las clases del paquete `juego.util` y el código parcial `juego.textui.Kamisado` se proporcionan en UBUVirtual.

- Completando una aplicación en modo gráfico.

- Generar la documentación correspondiente al código en formato HTML.
- Comprobar la completitud de la documentación generada previamente.
- Utilizar y enlazar con bibliotecas de terceros (i.e. biblioteca gráfica o *framework* de pruebas unitarias JUnit).
- Aportar los *scripts* correspondientes para realizar el proceso completo de compilación, documentación y ejecución (tanto en modo texto, como gráfico).

3. Descripción

A continuación se desglosan los distintos diagramas y comentarios, a tener en cuenta de cara a la implementación de la práctica. Algunos métodos de consulta (Ej: `obtener...`, `consultar...`, `esta...`, `contar...`, `tiene...`, etc.) y asignación (Ej: `colocar...`, `establecer…`, etc.) que no conllevan ningún proceso adicional, salvo la lectura o escritura de atributos, no se comentan, por motivos de brevedad. Es obligado consultar los diagramas de clases correspondientes, donde se muestran los conjuntos completos de métodos a implementar en cada clase.

3.1 Paquete `juego.modelo`

El paquete está formado por una interfaz, una clases abstracta, cuatro clases concretas y dos enumeraciones. En la Ilustración 2 se muestra el diagrama de clases correspondiente con el conjunto de métodos mínimo a implementar.

Comentarios respecto a la clase `Celda`:

- Inicialmente toda celda estará vacía. Solo contiene las coordenadas con su posición en el tablero y el color asignado.
- El método `eliminarTorre` elimina la torre de una celda previamente asignada.
- El método `establecerTorre` asigna la torre a la celda actual.
- El método `estaVacia` consulta si la celda tiene o no torre asignada.
- El método `obtenerColorDeTorre` obtiene el color de la torre asignada, o `null` si está vacía.
- El método `obtenerTurnoDeTorre` obtiene el turno de la torre asignada, o `null` si está vacía.
- El método `tieneCoordenadasIguales` devuelve `true` si la celda pasada como argumento tiene iguales coordenadas a la celda actual, o `false` en caso contrario, con independencia del resto de su estado.
- El método `toString` devuelve en formato texto el estado actual de la celda, con el siguiente formato: `"[X][Y] Color: C Turno: T Torre: C"`. Donde `X` es el n.º de fila, `Y` el n.º de columna, `C` es un color, `T` es el turno. Por ejemplo: `"[4][2] Color: V Turno: B Torre: N"` para la celda en las coordenadas del array [4][2], con color verde y con una torre colocada del turno blanco con color naranja. Si dicha celda estuviese vacía, el texto generado hubiese sido `"[4][2] Color: V Turno: - Torre: -"`.

Comentarios respecto a la enumeración `Color`:

- Contiene 8 valores: `AMARILLO`, `AZUL`, `MARRON`, `NARANJA`, `PURPURA`, `ROJO`, `ROSA` y `VERDE`.
- Permite almacenar y consultar el correspondiente carácter asociado a cada color: `'A'`, `'Z'`, `'M'`, `'N'`, `'P'`, `'R'`, `'S'` y `'V'` respectivamente a los valores previos.

- El método estático de utilidad `obtenerColorAleatorio` genera un color aleatoriamente de entre los ocho disponibles en la enumeración.

Comentarios respecto a la clase `Tablero`:

- El conjunto de celdas de un tablero debe **implementarse ahora** utilizando la **versiones genéricas de la interfaz `java.util.List<E>` y la clase concreta `java.util.ArrayList<E>`** (en **ningún caso** se utilizarán en esta práctica un *array* de celdas de dos dimensiones). Es conveniente acordarse que una tipo genérico se puede definir en función de si mismo como se ha visto en el Tema 5 (Ej: `Pila<Pila<Circulo>>`). Al instanciar un tablero se crean y asignan las correspondientes celdas vacías, con sus correspondientes coordenadas.
- El método `buscarCeldaOrigen(Turno, Color)` devuelve la celda de dicho color, en la fila de origen inicial del turno indicado (primera fila para turno blanco y última fila para turno negro).
- El método `buscarTorre` obtiene la celda que contiene la torre del turno y color indicada.
- El método `colocar(Torre, Celda)` coloca la torre en la celda indicada (método sobrecargado). Si las coordenadas de la celda no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `colocar(Torre, int, int)` coloca la torre en la fila y columna indicada. Si las coordenadas no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `colocar(Torre, String)` coloca la torre en la celda indicada en notación algebraica. Si las coordenadas de la celda, una vez traducido el texto, no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
    - Nota: todos los métodos sobrecargados `colocar` deben realizar el doble enganche entre torre y celda.
- El método `estaEnTablero` devuelve `true` si las coordenadas están en el tablero, `false` en caso contrario.
- El método `estanVaciasCeldasEntre` devuelve `true` si las celdas entre el origen y destino no contienen torres, es decir están vacías, o `false` en caso contrario Si las dos celdas están consecutivas sin celdas entre medias se devuelve `true`. Si las celda origen y destino no están alineadas en alguno de los sentidos definidos en la enumeración `Sentido`, se devuelve `false`. No se tiene en cuenta el estado de las celdas origen y destino, solo el de las celdas entre medias para comprobar si hay torres. Si las coordenadas de la celda origen o destino no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `hayTorreColorContrario(Turno)` comprueba si hay una torre de dicho turno, en la fila de origen o salida del turno contrario. Coincide con la condición de victoria en una ronda.
- El método `moverTorre` mueve la torre de la celda origen a destino. Si no hay torre en origen, o la celda destino no está vacía, no se hace nada. Si las coordenadas de la celda origen o destino no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerCelda`, devuelve la referencia a la celda del tablero. Si las coordenadas de la celda no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerCeldaDestinoEnJugada`, devuelve la referencia a la celda destino en la jugada introducida en notación algebraica (e.g. con `"a1c3"` retorna la celda en `[5][2]`). Si el formato de texto es incorrecto o las coordenadas de la celdas no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerCeldaOrigenEnJugada`, devuelve la referencia a la celda origen en la jugada introducida en notación algebraica (e.g. con `"a1c3"` retorna la celda en `[7][0]`). Si el formato de texto es incorrecto o las coordenadas de la celda no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**

- El método `obtenerCeldaParaNotacionAlgebraica`, devuelve la referencia a la celda en notación algebraica (e.g. con `"a1"` retorna la celda `[7][0]`). Si el formato de texto es incorrecto o las coordenadas de las celdas no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerCeldas` devuelve un **array de una dimensión** con todas las celdas del tablero. Se recorren las celdas de arriba a abajo, y de izquierda a derecha. Este método se puede utilizar en bucles *for-each* para recorrer todas las celdas del tablero de forma simplificada.
- El método `obtenerCoordenadasEnNotacionAlgebraica`, devuelve el texto correspondiente a la celda en notación algebraica (e.g. con la celda `[7][0]` se retorna `"a1"`). Si las coordenadas de la celda no están dentro del tablero **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerDistancia(Celda, Celda)` devuelve la distancia en celdas de origen a destino, sin incluir origen (e.g de la celda "a8" a la celda "d5" la distancia es 3). Si las coordenadas de alguna de la celdas no están dentro del tablero, **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerJugadaEnNotacionAlgebraica`, devuelve el texto correspondiente a un par de celdas origen y destino en notación algebraica (e.g. con la celda origen `[7][0]` y la celda destino `[5][2]` se retorna `"a1c3"`). Si las coordenadas de alguna de las celdas no están dentro del tablero, **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `obtenerNumeroTorres(Color)` devuelve el número de torres de un determinado color en el tablero (método sobrecargado).
- El método `obtenerNumeroTorres(Turno)` devuelve el número de torres de un determinado turno en el tablero (método sobrecargado).
- El método `obtenerSentido` obtiene el sentido de movimiento desde una celda origen a destino. Si las celda origen y destino no están alineadas en alguno de los sentidos definidos en la enumeración `Sentido`, se devuelve `null`. Si las coordenadas de alguna de las celdas no están dentro del tablero, **lanza una excepción `CoordenadasIncorrectasException`.**
- El método `toString` devuelve el estado actual del tablero en formato cadena de texto tal y como se mostraría a un jugador en plena partida. Ej: se muestra el tablero tras realizar algún movimiento de torres. En cada celda se indica en sus cuatros esquinas la letra con su color y en el centro, el turno y color de la torre (e.g. `"BN"` para turno blanco con torre verde, `"NM"` para turno negro con torre naranja, etc.) o bien un par de guiones si está vacía. Si es una **torre sumo uno**, se añade dicho número a la derecha (eg. `"BN1"`, `"NM1"`, etc.). Se muestra un ejemplo:

```
   a       b       c       d       e       f       g       h

   N..N    Z..Z    P..P    S..S    A..A    R..R    V..V    M..M
8  -BN1    -BZ-    -BP-    ----    -BA-    -BR-    ----    -BM-
   N..N    Z..Z    P..P    S..S    A..A    R..R    V..V    M..M

   R..R    N..N    S..S    V..V    Z..Z    A..A    M..M    P..P
7  ----    ----    ----    ----    ----    ----    -BV-    ----
   R..R    N..N    S..S    V..V    Z..Z    A..A    M..M    P..P

   V..V    S..S    N..N    R..R    P..P    M..M    A..A    Z..Z
6  ----    ----    ----    ----    ----    ----    ----    ----
   V..V    S..S    N..N    R..R    P..P    M..M    A..A    Z..Z

   S..S    P..P    Z..Z    N..N    M..M    V..V    R..R    A..A
5  -NM1    ----    ----    ----    ----    ----    ----    ----
   S..S    P..P    Z..Z    N..N    M..M    V..V    R..R    A..A

   A..A    R..R    V..V    M..M    N..N    Z..Z    P..P    S..S
4  ----    ----    -NR-    ----    ----    ----    ----    ----
   A..A    R..R    V..V    M..M    N..N    Z..Z    P..P    S..S

   Z..Z    A..A    M..M    P..P    R..R    N..N    S..S    V..V
3  ----    ----    ----    ----    ----    ----    ----    ----
   Z..Z    A..A    M..M    P..P    R..R    N..N    S..S    V..V

   P..P    M..M    A..A    Z..Z    V..V    S..S    N..N    R..R
2  ----    ----    ----    -BS-    ----    ----    ----    ----
   P..P    M..M    A..A    Z..Z    V..V    S..S    N..N    R..R

   M..M    V..V    R..R    A..A    S..S    P..P    Z..Z    N..N
1  ----    -NV-    ----    -NA-    -NS-    -NP-    -NZ-    -NN-
   M..M    V..V    R..R    A..A    S..S    P..P    Z..Z    N..N
```


Comentarios respecto a la interface `Torre`:

- **Este fichero se proporciona ya resuelto en UBUVirtual. No puede modificarse.**
- **Revisar los comentarios javadoc con la explicación de cada método.**

Comentarios respecto a la clase abstracta `TorreAbstracta`:

- Una torre se crea con un turno y color que no cambia a lo largo de la partida. Inicialmente una torre no está "colocada" sobre el tablero y no tendrá celda asignada.
- El método `establecerCelda`, asigna la celda a torre actual.
- El método `toString` devuelve en formato texto el estado actual de la torre, con el siguiente formato: `"TC"`. Donde `T` es el turno y `C` el color. Por ejemplo, para la torre blanca de color rosa, tendríamos `"BS"`.

Comentarios respecto a la clase concreta `TorreSimple`:

- Una **torre simple** se crea con un turno y color, que no cambia a lo largo de la partida. Inicialmente una torre no está "colocada" sobre el tablero y no tendrá celda asignada.
- El método `obtenerNumeroDientes` devuelve un valor de 0.
- El método `obtenerNumeroPuntos` devuelve un valor de 1.
- El método `obtenerMaximoAlcance` devuelve un valor `Integer.MAX_VALUE` (valor máximo de un entero, no hay límite en la distancia en celdas a desplazar una torre simple).
- El método `obtenerNumeroMaximoTorresAEmpujar` devuelve un valor de 0.
- El método `toString` devuelve en formato texto el estado actual de la torre, con el siguiente formato: `"TC"`. Donde `T` es el turno y `C` el color. Por ejemplo, para la torre blanca de color rosa, tendríamos `"BS"`.

Comentarios respecto a la clase concreta `TorreSumoUno`:

- Una **torre sumo uno** se crea con un turno y color, que no cambia a lo largo de la partida. Una torre sumo se instanciará siempre como resultado de la transformación de una torre simple que ha alcanzado la fila de salida del turno contrario.
- El método `obtenerNumeroDientes` devuelve un valor de 1.
- El método `obtenerNumeroPuntos` devuelve un valor de 3.
- El método `obtenerMaximoAlcance` devuelve un valor 5 (valor máximo de distancia en celdas a desplazar una torre sumo uno).
- El método `obtenerNumeroMaximoTorresAEmpujar` devuelve un valor de 1.
- El método `toString` devuelve en formato texto el estado actual de la torre sumo uno, con el siguiente formato: `"TCN"`. Donde `T` es el turno, `C` el color y `N` es el numero de dientes de dragón (siempre 1 en una torre sumo uno). Por ejemplo, para la torre sumo uno blanca de color rosa, tendríamos `"BS1"`.

Comentarios respecto a la enumeración `Turno`:

- Contiene los dos turnos en la partida: `BLANCO` y `NEGRO`.
- Permite almacenar y consultar el correspondiente carácter asociado a cada turno: `'B'` y `'N'` respectivamente a los valores previos.

A continuación se muestra el diagrama de clases completo para el paquete `juego.modelo`[3]. Las cláusulas **`throws`** no se detallan en el diagrama, pero están detalladas en la descripción de las clases del enunciado.

3 El símbolo ~ se traduce como acceso amigable.

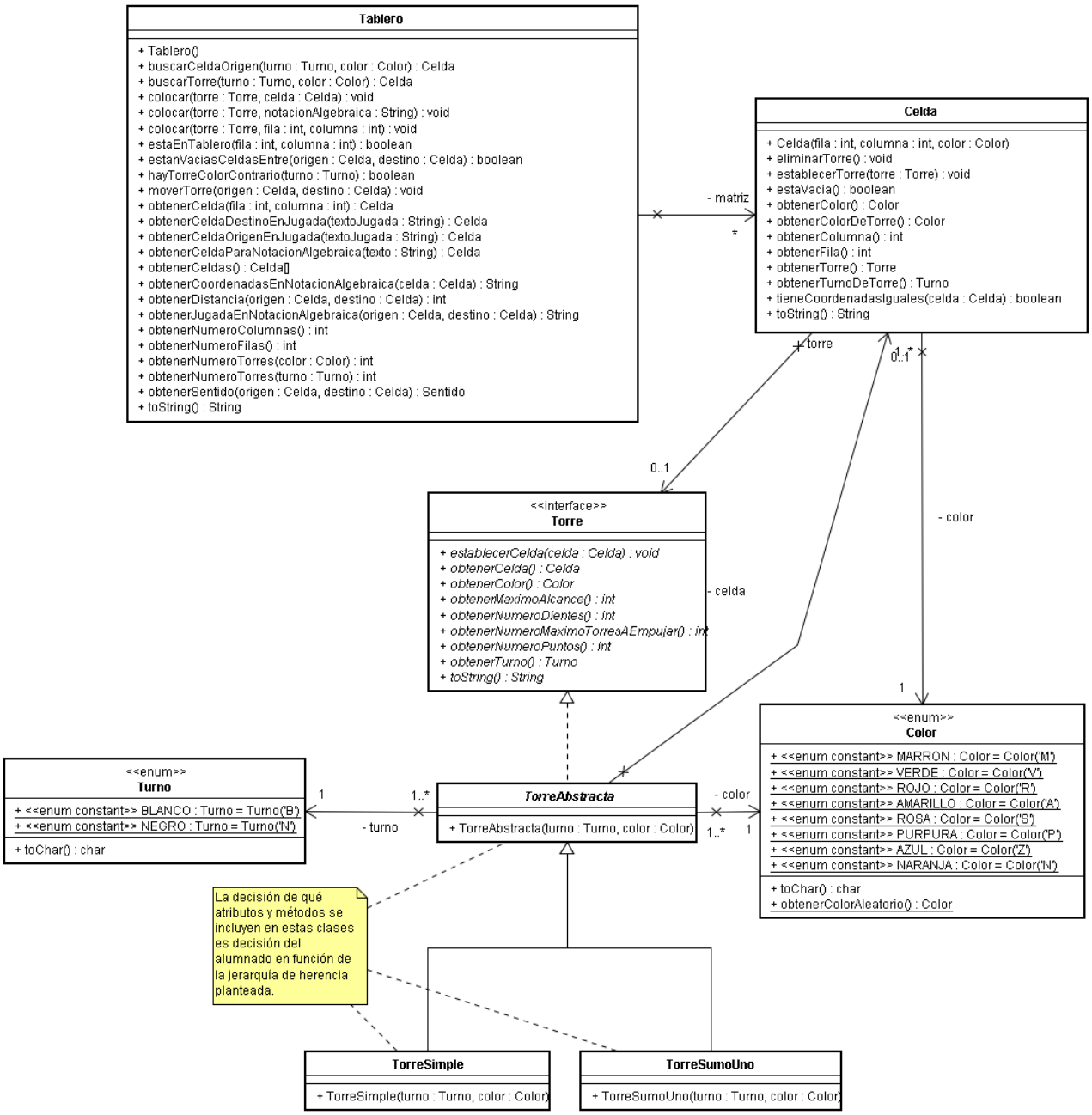


Ilustración 2: Modelo de clases del paquete juego.modelo

Dada la complejidad de algún método en la clase en `Tablero`, y en previsión de generar código duplicado, se recomienda **dividir el código de métodos largos, en métodos privados más pequeños y reutilizarlos, siempre que sea posible.** En algunos casos, la implementación de los métodos reutiliza a otros, por lo que se evita duplicar código. Algunos de los métodos propuestos son solo para su uso exclusivo en los tests automáticos, para verificar el correcto estado de los objetos.

3.2 Paquete `juego.control`

El paquete contiene una *interface*, una clase abstracta y dos clases concretas. Define la lógica de negocio o reglas del juego a implementar, comprobando la legalidad de las jugadas, realizando los movimientos y/o empujones (si son legales), gestionando el cambio de turno, la promoción de torres simples a torres sumo uno, la actualización del los últimos colores de celda a los que ha movido cada jugador y la gestión de finalización y reinicio de rondas y partida (ver Ilustración 3), dependiendo de tipo de árbitro seleccionado.

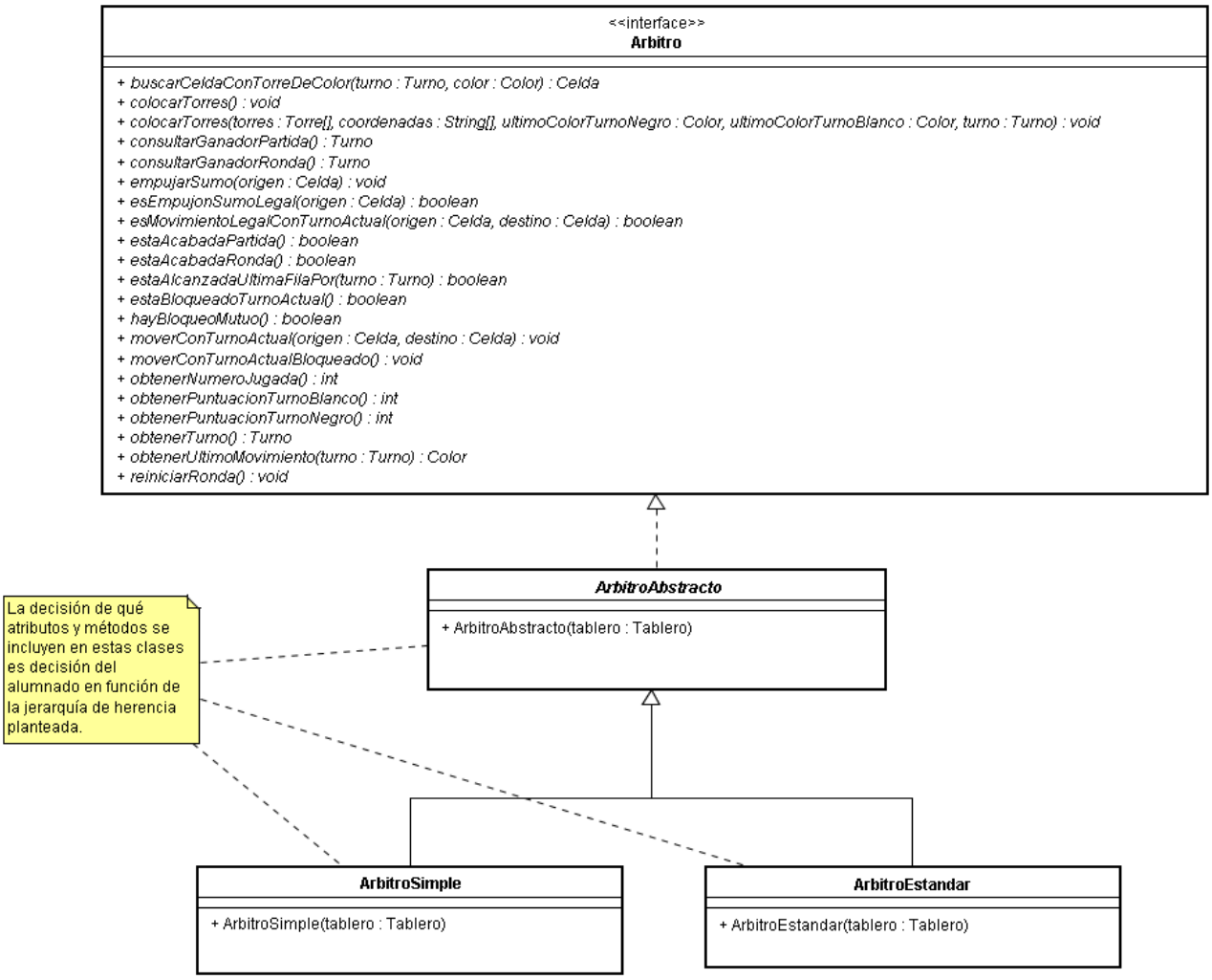


Ilustración 3: Modelo de clases del paquete juego.control

Comentarios respecto a la interface `Arbitro`:

- **Este fichero se proporciona ya resuelto en UBUVirtual. No puede modificarse.**
- **Revisar los comentarios javadoc con la explicación de cada método y las excepciones a lanzar en cada caso.**
- **Se ha añadido un método nuevo respecto a versiones previas:**
  - `boolean estaAlcanzadaUltimaFilaPor(Turno turno);`

Comentarios respecto a la clase abstracta `ArbitroAbstracto`:

- El constructor asigna el tablero con el que se va a jugar.

- Una clase abstracta puede contener atributos y redefinir los métodos que se crean convenientes de la interface.

Comentarios respecto a la clase concreta `ArbitroSimple`:

- El constructor asigna el tablero con el que se va a jugar.
- El método `consultarGanadorRonda` coincidirá en valor con el ganador de la partida, puesto que una partida simple solo tiene una ronda.
- El método `esEmpujonSumoLegal` devuelve siempre `false` puesto que en una partida simple no se pueden producir empujones sumo.
- El método `estaAcabadaRonda` devuelve `true` si está finalizada la única ronda que compone la partida o `false` en caso contrario. Dado que solo hay una ronda, en una partida simple, es equivalente a invocar `estaAcabadaPartida`.
- El método `empujarSumo` siempre lanza una excepción no comprobable `java.lang.UnsupportedOperationException`, dado que no debería invocarse sobre este tipo de árbitro.
- El método `reiniciarRonda` siempre lanza una excepción no comprobable `java.lang.UnsupportedOperationException`, dado que no debería invocarse sobre este tipo de árbitro.

Comentarios respecto a la clase abstracta `ArbitroEstandar`:

- El constructor asigna el tablero con el que se va a jugar.
- Este árbitro sí debe permitir transformar torres simples a torres sumo uno y realizar empujones sumo siempre que sean legales.
- Una partida tendrá varias rondas dependiendo de los puntos obtenidos por ambos turnos.
- En este caso, no hay métodos que lancen excepciones `java.lang.UnsupportedOperationException`.

Se omiten las relaciones de las clases del paquete control y modelo, deducibles a partir del enunciado y descripción de los métodos. Es decisión del alumnado implementar los distintos métodos en las clases que componen la jerarquía de herencia, decidiendo en qué clase es más adecuado redefinir el método. Se recuerda que pueden añadir métodos **`protected`** en las clases, siempre de manera justificada.

3.3 Paquete `juego.util`

Contiene una clase excepción y una enumeración (ver Ilustración 4).

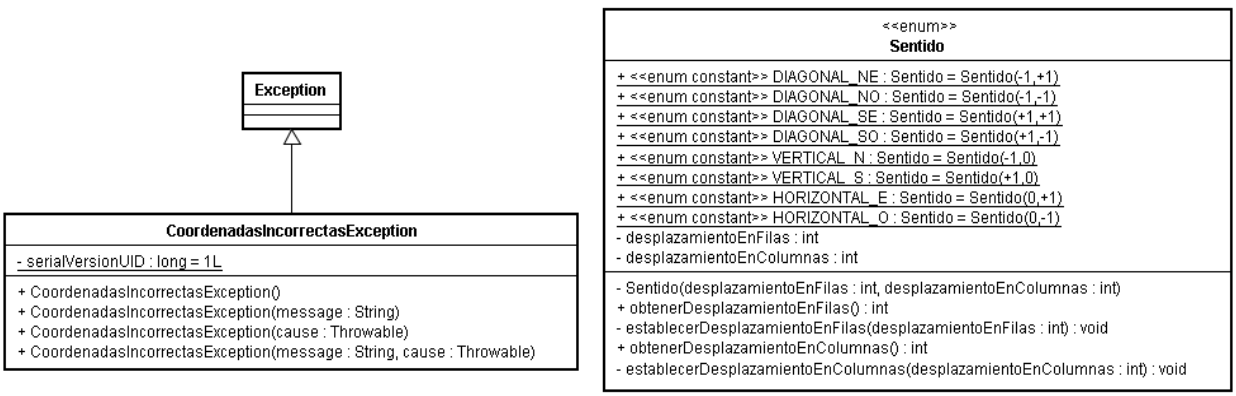


Ilustración 4: Modelo de clases del paquete `juego.util`

La excepción `CoordenadasIncorrectasException` es una **excepción comprobable, que obligatoriamente debe heredar de `java.lang.Exception`**. Incorpora los cuatro constructores, mostrados en la Ilustración 4 .

La enumeración `Sentido` contiene los ocho **sentidos de movimiento clásicos**, en la dirección horizontal, vertical o diagonal, junto con su desplazamiento en filas y columnas correspondiente a cada caso.

Se recuerda el uso del método `values` para obtener un *array* con todos los valores definidos en el tipo enumerado para simplificar el código. Es muy importante utilizar esta enumeración con los correspondientes valores de desplazamiento, para simplificar el código.

El fichero `Sentido.java`, se proporciona ya resuelto en UBUVirtual.

3.4 Paquete `juego.textui`

En este paquete se implementa, la interfaz en modo texto, que reutiliza los paquetes anteriores. Las clase raíz del sistema es `juego.textui.Kamisado`.

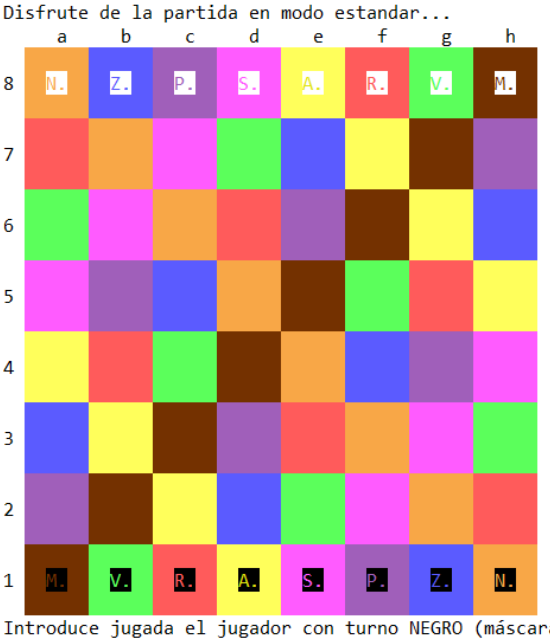
A la hora de ejecutar se puede pasar opcionalmente un argumento de texto que puede ser `simple` o `estandar`. En función del argumento pasado, se jugará con el tipo de árbitro correspondiente (a una o varias rondas respectivamente). Si no se pasa argumento, se juega en modo `simple` por defecto.

Un ejemplo de sintaxis de invocación sería (sin detallar cómo debe configurarse el `classpath`, eso es tarea del alumnado):

```
$> java juego.textui.Kamisado estandar
```

La salida en pantalla debe ser similar a la siguiente, tomando como caso de estudio una partida con árbitro estándar (las capturas se realizan en Eclipse, en consola los colores de fondo y letras se alternarían a negro y blanco respectivamente):

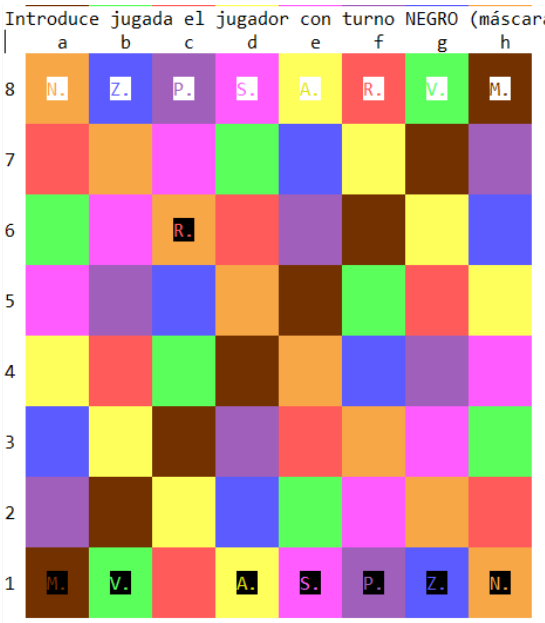
```
Bienvenido al juego del Kamisado
Para interrumpir partida introduzca "salir".
Para mostrar el estado del tablero en formato texto introduzca "texto".
Disfrute de la partida en modo estandar...
```




```
Introduce jugada el jugador con turno NEGRO (máscara cfcf para mover o cf para empujón sumo):
```


Suponiendo que en la primera jugada se juega de la celda `[7][2]` a `[2][2]` , torre de turno negro con color rojo a celda de color naranja en su vertical, introduciríamos la siguiente jugada `c1c6` y tendríamos la siguiente salida en pantalla:

```
Introduce jugada el jugador con turno NEGRO (máscara cfcf para mover o cf para empujón sumo): c1c6
```

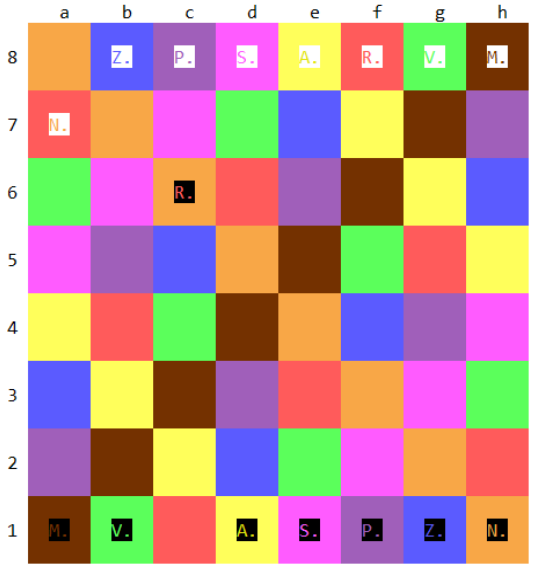



```
Último color de turno negro:NARANJA

Puntos de turno negro: 0        Puntos de turno blanco: 0
Introduce jugada el jugador con turno BLANCO (máscara cfcf para mover o cf para empujón sumo):
```

A continuación debe mover el jugador de turno blanco su torre de color naranja. Se mueve la celda `[0][0]` a `[1][0]` , torre de turno blanco con color naranja a celda de color rojo en su vertical. Introduciríamos la jugada `a8a7` y tendríamos la siguiente salida en pantalla:

```
Introduce jugada el jugador con turno BLANCO (máscara cfcf para mover o cf para empujón sumo): a8a7
```

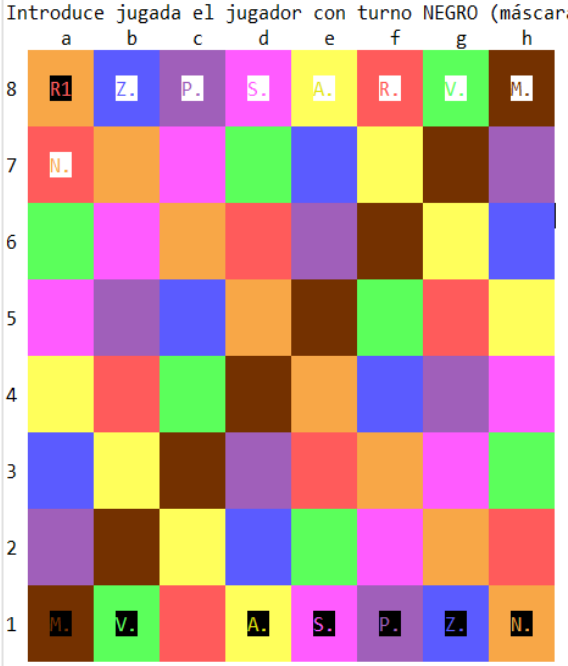



```
Último color de turno negro:NARANJA
Último color de turno blanco:ROJO

Puntos de turno negro: 0        Puntos de turno blanco: 0
Introduce jugada el jugador con turno NEGRO (máscara cfcf para mover o cf para empujón sumo):
```

A continuación debe mover el jugador de turno negro su torre de color rojo. Se mueve la celda `[2][2]` a `[0][0]` , torre de turno negro con color rojo a celda de color naranja en su diagonal. Introduciríamos la siguiente jugada `c6a8` y tendríamos la siguiente salida en pantalla, finalizando la ronda puesto que hemos alcanzado la fila del jugador contrario, **reemplazando** la **torre simple** por una **torre sumo uno**, reiniciando la ronda y actualizando puntos de ambos turnos:

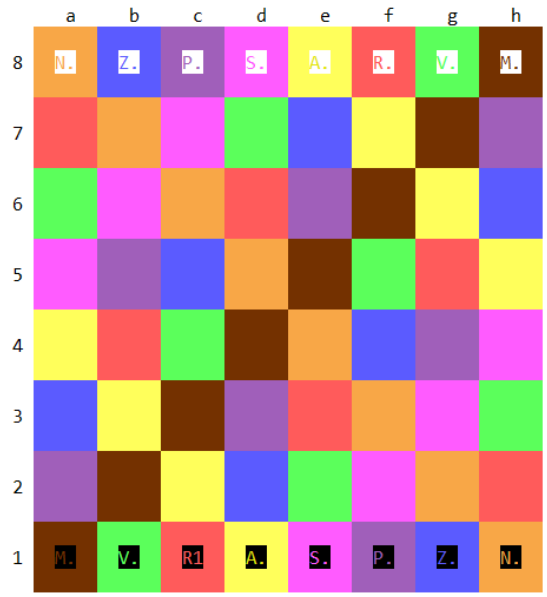
```
Introduce jugada el jugador con turno NEGRO (máscara cfcf para mover o cf para empujón sumo): c6a8
```




```
Último color de turno negro:NARANJA
Último color de turno blanco:ROJO

Puntos de turno negro: 1        Puntos de turno blanco: 0
Reiniciamos ronda...
```

A continuación se mostrará la **siguiente ronda**, con las **torres colocadas en su posición de partida original**, con el único cambio de que **tenemos una nueva torre sumo uno** roja del jugador con turno negro. Inicia la siguiente ronda el turno que perdió la ronda previa, en este ejemplo el turno blanco.




```
Introduce jugada el jugador con turno BLANCO (máscara cfcf para mover o cf para empujón sumo):
```

Así se continua jugando, hasta que alguno de los dos jugadores **alcanza o supera una puntuación de 3 puntos**, (bien porque tiene 3 torres sumo uno o bien porque ha alcanzado con una torre sumo uno la fila de origen del turno contrario).

Si la jugada introducida **no es legal,** se debe informar del error al usuario, solicitando de nuevo que introduzca la jugada y sin saltar el turno. No hay límite en el número de reintentos.

Si la jugada provoca un **bloqueo** en el jugador contrario, se informará, realizando el movimiento de "distancia cero", cambiando el turno y actualizando los colores de últimos movimientos, como se realizó en la anterior práctica.

Si la jugada provoca un **bloqueo mutuo** se finaliza la ronda o la partida informando del ganador, con los mismos criterios aplicados en la anterior práctica.

En caso de **victoria**, se mostrará siempre el estado del tablero final y se indica qué jugador ha ganado.

Si en algún momento el usuario introduce `"texto"` (ignorando mayúsculas o minúsculas) se mostrará el tablero en formato texto (correspondiente al texto generado con el método `toString` del `Tablero`), y se vuelve a preguntar por la jugada.

Si en algún momento el usuario introduce "`salir`" se interrumpe la partida y se finaliza simplemente mostrando en pantalla: `Interrumpida la partida, se concluye el juego.`

Este fichero se proporciona parcialmente resuelto en UBUVirtual. Solo hay que completar el método `main`, reutilizando el resto de métodos proporcionados, sin modificarlos, para contruir el "algoritmo".

3.5 Paquete `juego.gui/juego.gui.images`

Estos paquetes implementan la interfaz gráfica del juego y se proporciona ya resuelta por los profesores. La clase raíz del sistema es `juego.gui.Kamisado`.

A la hora de ejecutar se puede pasar opcionalmente un argumento de texto que puede ser `simple` o `estandar`.

En función del argumento pasado se jugará con el tipo de árbitro correspondiente (a una o varias rondas respectivamente). Si no se pasa argumento se juega en modo `simple`.

Ejemplo de invocación (sin detallar cómo debe configurarse el `classpath` quedando como ejercicio para el alumnado) en modo estandar :

```
$> java juego.gui.Kamisado estandar
```

La interfaz inicial será similar a la mostrada en la Ilustración 5. Por motivos de brevedad, solo se muestra la interfaz en modo estándar, siendo equivalente en modo simple.

En la parte superior se muestra el turno actual (e.g. NEGRO), y los últimos colores de cada turno (que inicialmente estarán sin definir). También se muestran las **puntuaciones de cada turno y un botón para realizar el empujón sumo**, una vez seleccionada una celda.

Se proporciona una casilla para activar/desactivar los *tooltips* de ayuda con el texto del estado de cada celda en pantalla (i.e. si el usuario tiene problemas para distinguir los colores, posicionando el ratón encima del elemento concreto, se muestra un texto descriptivo).

El botón de reiniciar, borra el estado actual, reiniciando una nueva partida.

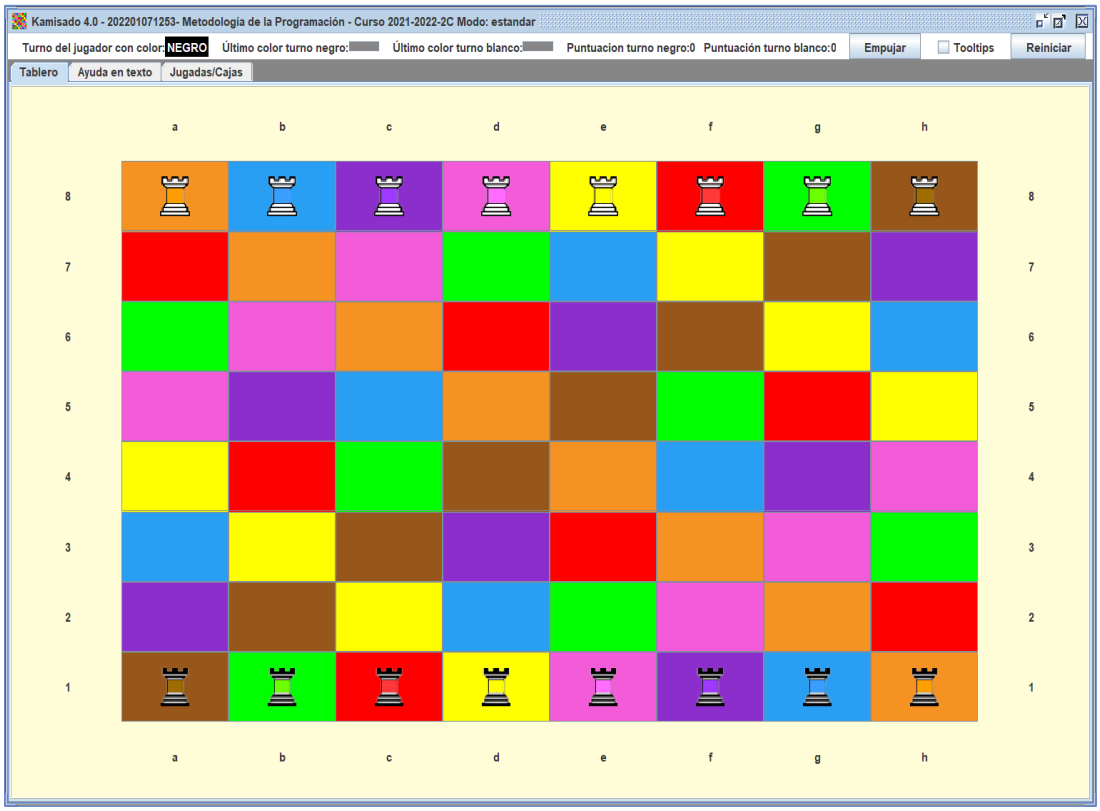


Ilustración 5: Pantalla de inicio en modo gráfico

Se proporcionan dos pestañas adicionales:

- ***Ayuda en texto*** donde se muestra el tablero actual en formato texto (equivalente al resultado del método `toString` de la clase `Tablero`)
- ***Jugadas*** donde se muestran las jugadas realizadas por cada jugador en notación algebraica.

Para realizar un **movimiento de desplazamiento de torre**, el jugador con turno actual debe seleccionar (hacer *click*) sobre la celda origen, y luego seleccionar la celda destino. Si el movimiento es ilegal se mostrará un diálogo con el error en pantalla. En caso contrario se realiza el movimiento, moviendo la torre y se actualiza la nueva información de turno y último color de su turno.

Para realizar un **movimiento de empujón sumo**, el jugador con turno actual debe seleccionar (hacer *click*) sobre la celda origen que empuja la contraria y pulsar el botón *Empujar*. Si el movimiento es ilegal se mostrará un diálogo con el error en pantalla. En caso contrario se realiza el empujón, moviendo la torre actual y la torre empujda una celda, y se actualiza la nueva información de turno y último color de su turno.

Cuando se alcanza la fila contraria, la partida finaliza indicando el ganador de la ronda o de la partida (según el tipo de partida) y actualizando puntuaciones, como se muestra en la Ilustración 6.

Ilustración 6: Ronda finalizada

Al finalizar la ronda la torre simple se transforma en una torre sumo uno, como se muestra en la Ilustración 7 (la torre roja del turno negro pasa a ser una torre sumo uno). El icono muestra un número 1 asignado a dicha torre. Las torres se vuelven a colocar como al inicio de la partida, pero reinicia la ronda el turno que acaba de perder.

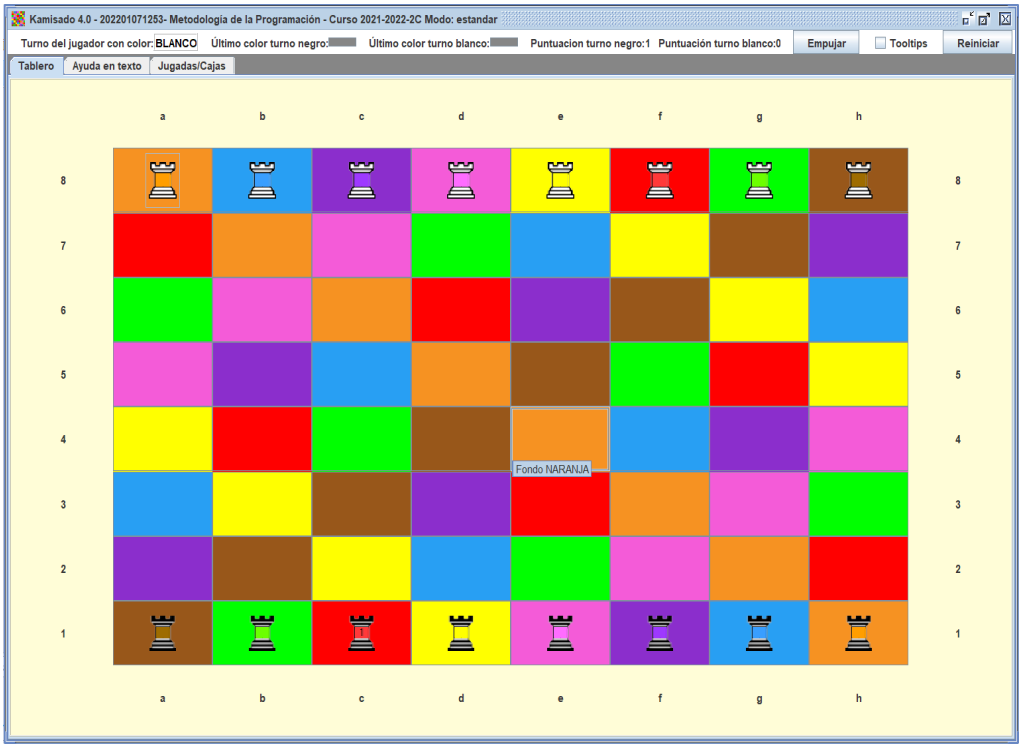


Ilustración 7: Renicio de ronda en partida estándar con nueva torres sumo uno.

Según avancen las rondas se puede dar la situación de que una torre sumo uno pueda realizar un empujón, como se muestra en la Ilustración 8, donde la torre sumo uno roja del turno negro puede empujar la torre púrpura del turno blanco (e.g. equivalente a introducir "c4" en modo texto).

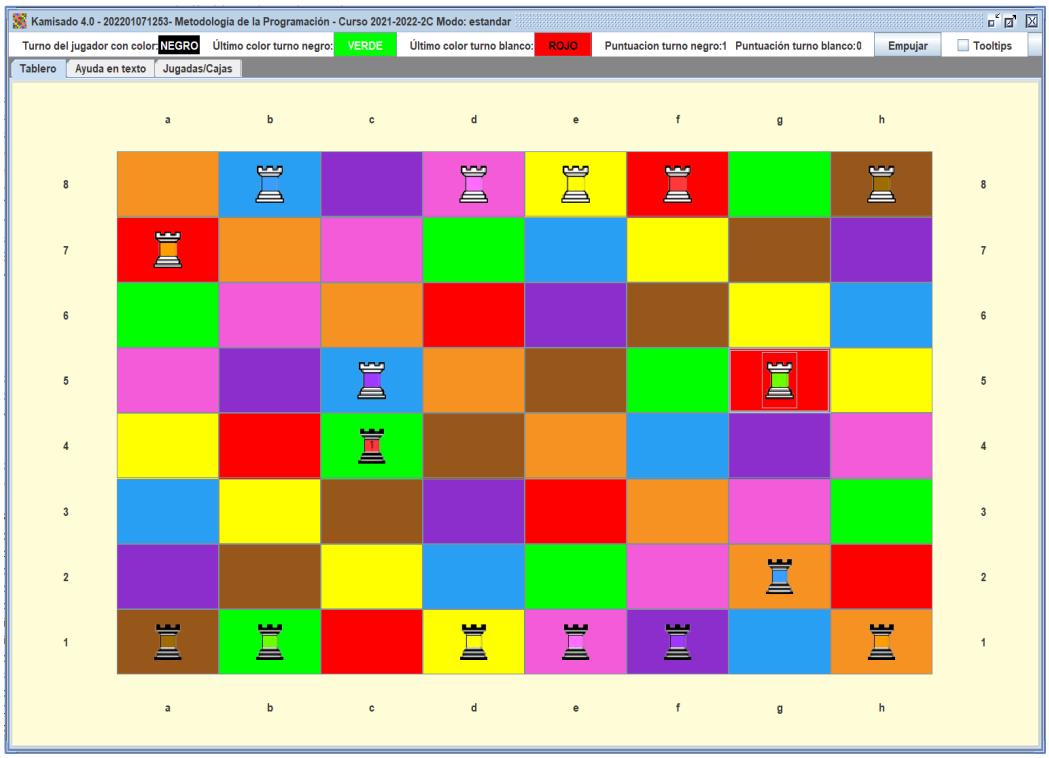


Ilustración 8: Situación de empujón sumo posible para torre sumo uno.

En la Ilustración 8, la torre sumo uno (de color rojo del turno negro) se selecciona y se pulsa el botón *Empujar*. Como resultado se empuja la torre púrpura del turno blanco, desplazando ambas torres tal y como se muestra en la Ilustración 9. Ambas torres se mueven verticalmente en sentido vertical norte en este ejemplo (recordemos que las torres sumo uno blancas empujan en sentido vertical sur).

El turno blanco, en este ejemplo, pierde turno al ser empujado, y el nuevo color de torre negra a mover se determina por el color de celda donde ha quedado la celda de turno blanca empujada (naranja en este ejemplo al quedar desplazada la torre púrpura de turno blanca a una celda naranja).

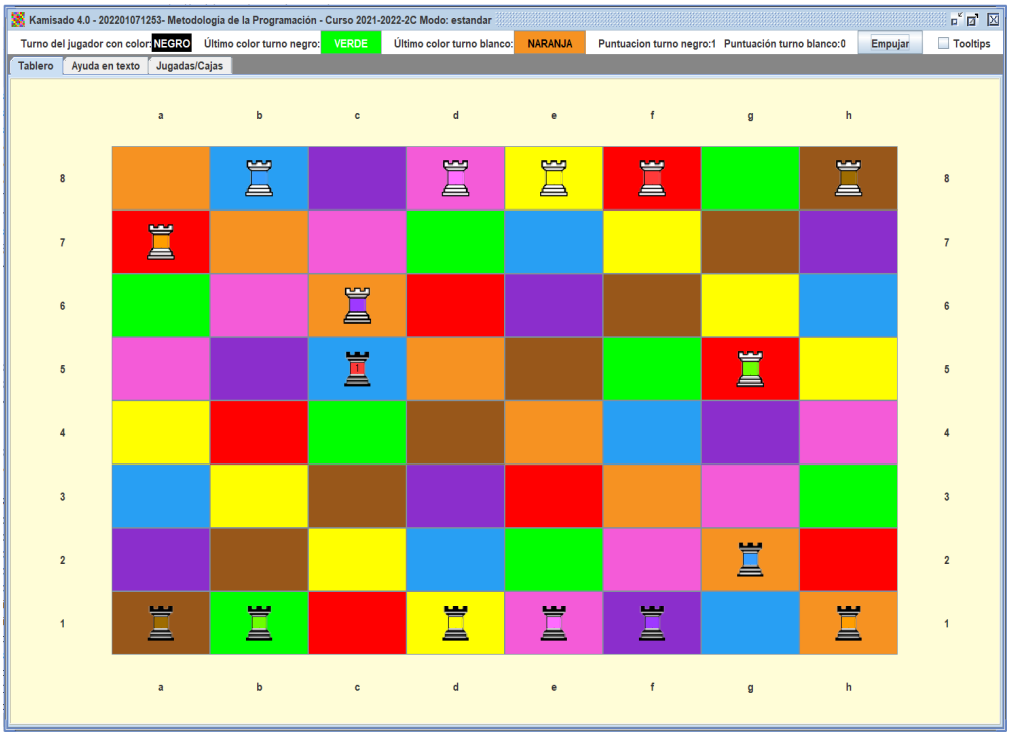


Ilustración 9: Resultado posterior a un empujón sumo

En ambas partidas (simple o estándar) si el movimiento genera un **bloqueo**, se informa al usuario, se realiza el movimiento de "distancia cero", cambiando el turno y actualizando el color de último movimiento, como ya se realizó en la anterior práctica.

Si en ambas partidas (simple o estándar) se produce un **bloqueo mutuo**, en primer lugar se informará del bloqueo del jugador con siguiente turno, y a continuación de la situación de bloqueo mutuo, actualizando el estado de la partida e informando en pantalla de la finalización de la ronda o partida y del ganador de la misma, como ya se realizó en la anterior práctica.

La **partida finaliza** cuando se alcanza la puntuación mínima según el tipo de partida (1 en simple, 3 en estándar). En la Ilustración 10, se muestra una partida estándar finalizada, utilizando la torre sumo uno para ganar en la segunda ronda, con una puntuación acumulada de 4 puntos.

Ilustración 10: Partida estándar finalizada con torre sumo uno llegando a fila contraria

Las clases correspondientes a estos paquetes con la interfaz gráfica **se proporcionan en formato binario en un fichero con nombre** `kamisado-gui-lib-4.0.0.jar` y se debe usar con los paquetes construidos previamente configurando correctamente el `classpath` y **sin descomprimir en ningún caso el fichero .jar**. No se utilizarán en estas prácticas módulos Java (**no debemos tener ningún ficheros** `module-info.java` **en nuestro proyecto).**

4. Entrega de la práctica

4.1 Fecha límite de entrega

- Ver fecha indicada en UBUVirtual de la tarea **[MP] Entrega de Práctica Obligatoria 2 - EPO2-2C.**

4.2 Formato de entrega

- Se enviará un fichero `.zip`. `.rar` o `.tar.gz` a través de la plataforma **UBUVirtual** completando la tarea **[MP] Entrega de Práctica Obligatoria 2 – EPO2-2C.**
- Los ficheros fuente estarán codificados **OBLIGATORIAMENTE** en formato **UTF-8** (Tip: comprobar en Eclipse, en *File/Properties* del proyecto que el valor *Text file encoding* está configurado a dicho valor).
- **TODOS** los ficheros fuente `.java` deben incluir los **nombres y apellidos de los autores** en su cabecera y deben estar **correctamente indentados. En caso contrario la calificación es cero.**
- El fichero comprimido seguirá alguno de los siguientes formatos de nombre **sin utilizar tildes**:
  - **`Nombre PrimerApellido-Nombre PrimerApellido.zip`**

  - **`Nombre PrimerApellido-Nombre PrimerApellido.rar`**
  - **`Nombre PrimerApellido-Nombre PrimerApellido.tar.gz`**
- Ej: si los alumnos son Estrella Morales y Raúl Marticorena, su fichero `.zip` se llamará sin utilizar tildes **`Estrella Morales-Raul Marticorena.zip`**.
- **Se puede realizar la práctica individualmente o por parejas. Se calificará con los mismos criterios en ambos casos. Si se hace por parejas, la nota de la práctica es la misma para ambos miembros.**
- **Aunque la práctica se haga por parejas, se enviará individualmente por parte de cada uno de los dos integrantes a través de UBUVirtual. Verificar que ambas entregas son iguales en contenido. En caso de NO coincidencia, se penalizará un 25% a ambos.**
- **NO se admiten envíos posteriores a la fecha y hora límite, ni a través de otro medio que no sea la entrega de la tarea en UBUVirtual. Si no se respetan las anteriores normas de envío la calificación es directamente cero.**
- **Cualquier situación de plagio detectado en las prácticas, conlleva la aplicación del reglamento de exámenes.**

Se deber entregar el **proyecto de Eclipse completo,** en el fichero comprimido solicitado.

En dicho proyecto deben existir los siguientes ficheros y directorios.

- `/leeme.txt`: fichero de texto, que contendrá los nombres y apellidos de los integrantes y las aclaraciones que los alumnos crean oportunas poner en conocimiento de los profesores.
- 🗀 `lib`: contiene las siguientes bibliotecas (descargar desde UBUVirtual)
  - `JColor-5.0.0.jar`: biblioteca utilizada para mostrar colores en consola texto.
  - `kamisado-gui-lib-4.0.0.jar`[*]: biblioteca con la interfaz gráfica **proporcionada por los profesores** para poder ejecutar la aplicación en modo gráfico.
  - `junit-platform-console-standalone-1.7.2.jar`: biblioteca utilizada para la ejecución de los tests con el *framework* JUnit 5.
- 🗀 `src`: ficheros fuentes (`.java`) y ficheros de datos necesarios para poder compilar el producto completo.
- 🗀 `test`: ficheros fuentes (`.java`) con los tests automáticos **proporcionados por el profesorado**.
- 🗀 `bin`: ficheros binarios (`.class`) generados al compilar.
- 🗀 `doc`: documentación HTML generada con **`javadoc`** de todos los ficheros fuentes.
- `/documentar.bat o /documentar.sh`: fichero de comandos con la invocación al generador de documentación `javadoc` para generar el contenido del directorio `doc` a partir de los ficheros fuente en la carpeta `src`.

Comprobar previamente siempre antes de la entrega el correcto contenido del fichero .zip y del proyecto entregado..

4.3 Comentarios adicionales

- **No se deben modificar los ficheros binarios ni los tests proporcionados.** En caso de ser necesario, por errores en el diseño/ implementación de los mismos, se notificará a los profesores de la asignatura quienes publicarán en UBUVirtual la corrección y/o modificación. Se modificará el número de versión del fichero en correspondencia con la fecha de modificación y se publicará un listado de erratas.

* Si se publica alguna corrección, el fichero cambiará en número de versión

- **Se requiere la utilización de los tests automáticos por parte de los alumnos** para verificar la corrección de la entrega realizada. Aunque los tests no aseguran al 100% la corrección de la solución dada (no son exhaustivos y se pueden pasar con soluciones no óptimas), aseguran un funcionamiento mínimo y la autocorrección de la solución aportada. **Como mínimo se piden pasar con éxito los tests denominados como "básicos".**
- **Se realizará un cuestionario individual para probar la autoría de la misma.** El peso es de 5% sobre la nota final con nota de corte 4 sobre 10 para superar la asignatura.

4.4 Criterios de valoración

- La práctica es obligatoria, entendiendo que su no presentación en la fecha marcada, supone una calificación de cero sobre el total de la práctica. La nota de corte en esta práctica es de 5 sobre 10 para poder superar la asignatura. Se recuerda que la práctica tiene un peso del **15% de la nota final de la asignatura**.
- Se valorará **negativamente** métodos con un **número de líneas grande** (>30 líneas) sin contar comentarios ni líneas en blanco, ni llaves de apertura o cierre, indentando el código con el formateo por defecto de Eclipse. En tales casos, se debe dividir el método en métodos privados más pequeños. Siempre se debe evitar en la medida de lo posible la repetición de código.
- No se admiten ficheros cuya estructura y contenido no se adapte a lo indicado, con una valoración de cero.
- No se corrigen prácticas que **no compilen**, ni que contengan **errores graves en ejecución** con una valoración de cero.
- No se admite **código no comentado** con la especificación para **javadoc** con una valoración de cero.
- No se admite no entregar la documentación **HTML** generada con **javadoc.**
- Se valorará negativamente el porcentaje de fallos en documentación al generarla con javadoc y al comprobarla con el *plugin* JAutodoc de Eclipse.
- **Es obligatorio seguir las convenciones de nombres vistas en teoría** en cuanto a los nombres de paquetes, clases, atributos y métodos.
- Se penalizarán los **errores en los tests automáticos.**
- Aun superando todos los tests, si es **imposible completar una partida, con un mínimo de jugadas que finalice en partida ganada,** tanto en el modo texto o gráfico, la calificación de la prática estará **penalizada un 40%.**
- Porcentajes/pesos[4] aproximados en la valoración de la práctica:

Apartado	Cuestiones a valorar	Porcentaje (Peso)
Cuestiones generales de funcionamiento	Integrada la solución con la interfaz gráfica (10%). Documentada sin errores (8%). Scripts correctos (1%). Utiliza package-info. Extras de documentación (1%)	10,00%
Kamisado (textui)	Corrección del algoritmo propuesto. Correcto funcionamiento. Correcto tratamiento de excepciones.	10,00%
Arbitro ArbitroAbstracto ArbitroSimple ArbitroEstandar	Uso de modificadores de acceso. Atributos correctos. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de cambio de estado correctos. Métodos de consulta correctos. Correcta gestión de movimientos, turno y de últimos colores. Reglas correctamente implementadas. Detección y resolución de bloqueo. Detección y resolución de bloqueo mutuo. Uso de constantes simbólicas. Correcta estructura de atributos y métodos en la jerarquía de herencia. Correctas partidas simples. Correctas partidas estándar con cambio de torre sumo uno. Correcto tratamiento de excepciones.	30,00%
Tablero	Uso de modificadores de acceso. Atributos correctos. Constructor correcto.	30,00%

4 Los porcentajes pueden varias ligeramente, al haber redondeado decimales.

	Correctas inicializaciones con torres. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de cambio de estado correctos. Métodos de consulta correctos. Uso de constantes simbólicas. Correcto manejo de la notación algebraica. Correcta incorporación de la genericidad. Correcto tratamiento de excepciones.	
Torre TorreSimple TorreSumoUno	Uso de modificadores de acceso. Atributos correctos. Cambios de estado correctos. Textos generados correctos. Métodos de cambio de estado. Correcta jerarquía de herencia de torres con redefiniciones correctas.	12,50%
Celda	Uso de modificadores de acceso. Atributos correctos. Cambios de estado correctos. Textos generados correctos. Métodos de cambio de estado.	2,50%
Color Turno	Correcta implementación de enumeraciones.	2,50%
CoordenadasIncorrectasException	Correcta implementación de la excepción.	2,50%

Anexo 1. Recomendaciones en el uso de la interfaz **`List`** y la clase **`java.util.ArrayList`**

La interfaz `java.util.List` es implementada por la clase `java.util.ArrayList`. Dicha clase implementa un *array* que **dinámicamente** puede cambiar sus elementos, aumentando o disminuyendo su tamaño. Recordad que en contraposición, los *arrays* en Java son constantes en tamaño una vez determinado.

En esta **segunda práctica** utilizaremos la versión **GENÉRICA tanto de la interfaz como de la clase concreta.**

Una vez instanciado un `ArrayList` (con cualquiera de los constructores que aporta), tendremos una lista vacía.

Cuando se añaden elementos se pueden utilizar los métodos:

`public boolean` `add(E e)`

Añade el elemento al final de la lista.

O bien:

`public void` `add(`**`int`**` index, E element)`

Inserta el elemento en la posición especificada. Desplaza el elemento en esa posición (si hay alguno) y todos los demás elementos a su derecha (añade 1 a sus índices)

Lanza una excepción – si el índice está fuera del rango (`index < 0 || index > size()`)

Mientras que el primer método añade siempre al final incrementando a su vez el tamaño (`size()`), el segundo permite añadir de forma indexada siempre que el índice **NO** esté fuera del rango (`index < 0 || index > size()`).

Si se quiere inicializar dimensionando de manera adecuada el número de elementos y se desconoce el valor a colocar en ese momento, se permite la inicialización con valores nulos (`null`) como se muestra en el ejemplo.

Ej:

```
// En este ejemplo se utiliza una variable de tipo interfaz para manejar el ArrayList.
// Siempre que sea posible se deben utilizar interfaces para manejar objetos concretos
List<Integer> array = new ArrayList<Integer>(10); // capacidad 10 pero tamaño inicial 0
System.out.println("Tamaño del array list:" + array.size()); // se muestra 0 en pantalla

// cualquier intento de realizar una invocación a add(index,element) con index != 0
// provocaría una excepción de tipo IndexOutOfBoundsException
for (int i = 0; i < 10; i++){
    array.add(null);
}
// tamaño de array (size()) es 10 a la finalización del bucle
```

Para modificar una posición determinada se utiliza el método:

`public E set(int index, E element)`

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para consultar el elemento en una posición determinada se utiliza el método:

`public E get(int index)`

Lanza una excepción IndexOutOfBoundsException – si el índice está fuera del rango (`index < 0 || index >= size()`)

Para añadir todos los elementos de otro `ArrayList`, al final del `ArrayList` actual se puede utilizar el método:

`public boolean addAll(Collection<? extends E> c)`

teniendo en cuenta que un `ArrayList` es una `Collection`.

Para extraer eliminando además el elemento del `ArrayList` se puede utilizar el método:

`public E remove(int index)`

Si queremos recorrer todos los elementos de un `List` con genericidad se puede utilizar un bucle `for-each`. Estos bucles facilitan el recorrido de estructuras iterables, avanzando automáticamente y llevando el control de finalización (sin embargo solo permiten recorrer en un sentido). Por ejemplo parar recorrer una lista de celdas y mostrar sus valores de fila y columna en pantalla:

```
public void mostrarCeldas(List<Celda> celdas) {
	for (Celda celda : celdas) {
		System.out.println(celda.obtenerFila() + "-" + celda.obtenerColumna());
	}
}
```

Para utilizar el resto de métodos (vaciar, consultar el número de elementos, etc.) se recomienda consultar la documentación en línea de la interface y de la clase, aunque el conjunto de métodos indicados debería ser suficiente para la resolución de la práctica.

Anexo 2. Programación defensiva y tratamiento de excepciones

En esta práctica se ha optado por un enfoque de **programación defensiva**, lanzando una excepción **comprobable** en todos aquellos métodos de `Arbitro` y `Tablero` que reciben celdas o traducen textos para obtener la celda correspondiente.

A la hora de resolver la práctica, se debe tener en cuenta que en algunos casos, al invocar dichos metódos desde otros métodos, pueden surgir excepciones realmente provocadas por errores del programador, y no por los datos de entrada externos al programa (e.g. coordenadas incorrectamente generadas por nuestros propios bucles, cuando recorremos el tablero).

En tales casos, dado que no podemos ni debemos modificar las signaturas de los métodos del enunciado añadiendo cláusulas **`throws`** con excepciones comprobables, habrá que revisar y tener en cuenta que esas excepciones **señalan realmente un error de programación interno** y que debería ser indicado con el lanzamiento de una **excepción no comprobable**, en nuestro caso `java.lang.RuntimeException` (revisad la teoría vista en el Tema 6. Programación defensiva y tratamiento de excepcioens).

Por lo tanto, en algunos casos, se sugiere revisar la adecuación del mecanismo de **captura** y **sustitución de excepción comprobable por no comprobable**, y siempre utilizando encadenamiento de excepción como se muestra en el siguiente ejemplo.

Ej:

```
try{
    // Código que genera lanzamiento de excepción
}
catch(CoordenadasIncorrectasException ex) {
    throw new RuntimeException("Error grave en el código.", ex);
}
```

Recursos

Bibliografía complementaria:

[Oracle, 2021] Lesson: Language Basics. The Java Tutorials (2021). Disponible en http://docs.oracle.com/javase/tutorial/java/.

[JDK16, 2021] JDK 16 Documentation. Disponible en https://docs.oracle.com/en/java/javase/16/

Wikipedia contributors. (2021, July 9). Kamisado. In *Wikipedia, The Free Encyclopedia*. Retrieved 10:39, September 18, 2021, from https://en.wikipedia.org/w/index.php?title=Kamisado&oldid=1032833866

Enlaces a relgas del juego y **Vídeos en YouTube**

Algunos enlaces y vídeos donde se revisan las normas del juego, aunque incluyen no solo partidas simples, sino **avanzadas** con **reglas más complejas.** Aunque puede haber **alguna diferencia con el juego simplificado que hemos planteado**, pero en su mayoría las reglas son coincidentes.

[Entretenimiento Digital, 2021] Reglas del juego Kamisado. Disponible en https://entretenimientodigital.net/reglas-del-juego-kamisado/

[Ketty, 2011] Kamisado [Juego de Mesa / BoardGame] (2011) Disponible en https://www.youtube.com/watch?v=WRovXsMKOsw

[Kamisado, 2019] Kamisado (2019). Disponible en https://www.youtube.com/watch?v=wD-OXFw8Lrs (en inglés).

[Kamisado, 2021] Kamisado: Cómo jugar + Opinión. (2021). Disponible en https://www.youtube.com/watch?v=8BxRE0qPrpg&t=3s

[Sribd Inc., 2021] Kamisado – Reglas en español. Disponible en https://es.scribd.com/document/390479524/Kamisado-Reglas-en-Espanol

Licencia

Autor: Raúl Marticorena & Estrella Morales

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2022

Este obra está bajo una licencia Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Unported. No se permite un uso comercial de esta obra ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula esta obra original

Licencia disponible en http://creativecommons.org/licenses/by-nc-sa/4.0/