Grado en Ingeniería de la Salud

Metodología de la Programación

Práctica Obligatoria 1 – Segunda Convocatoria

Kamisado 3.0

Docentes:

Raúl Marticorena

Estrella Morales

Índice de contenidos

1. Introducción

El objetivo fundamental es implementar una variante **simplificada** del juego **Kamisado.** A continuación se describen las reglas del juego.

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 (ver Ilustración 1):

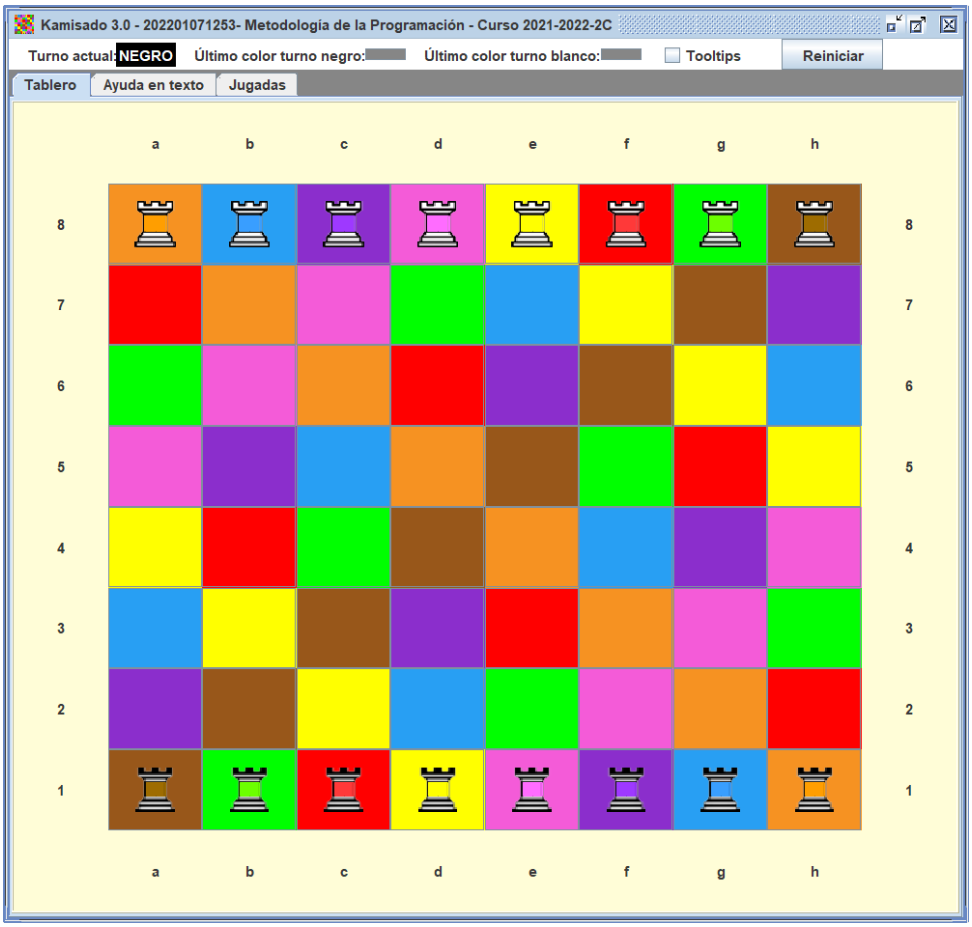


Ilustración 1: Interfaz gráfico de inicio de partida

1 Denominadas también "torres dragón".

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. 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 **finaliza** cuando uno de los jugadores consigue **colocar una de sus torres en la fila de salida del turno contrario**.

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 previas), 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".

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`
	`Turno`
		`3 clases`
		`2 enumeraciones`
			Modelo fundamental.

`juego.control`	`Arbitro`	`1 clase`	Lógica de negocio (reglas del juego).
`juego.util`	**`Sentido`**	`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 (`private`), 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.
  - 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).

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

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. 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 tres clases 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`:

- Un tablero se considera como un conjunto de celdas, cada una en una posición (fila,columna). Suponiendo que el tablero es de 8 filas x 8 columnas, entonces tenemos: [0][0] las coordenadas de la esquina superior izquierda, [0][7] las coordenadas de la esquina superior derecha, [7][0] las coordenadas de la esquina inferior izquierda y [7][7] las coordenadas de la esquina inferior derecha. Se numera de izquierda a derecha y en sentido descendente.
- El conjunto de celdas de un tablero **debe implementarse con un *array*[3] de celdas de dos dimensiones**. Al instanciar un tablero se crean y asignan las correspondientes celdas vacías, con sus correspondientes coordenadas y colores.
- 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).
- El método `colocar(Torre, int, int)` coloca la torre en la fila y columna indicada. Si los valores de fila y columna son incorrectos (no están en los límites del tablero), no se hace nada (método sobrecargado).
- El método `colocar(Torre, String)` coloca la torre en la celda indicada en notación algebraica. Si la celda en notación algebraica es incorrecta en sintaxis, no se hace nada (método sobrecargado).
    - Nota: todos los métodos sobrecargados `colocar` deben realizar el doble enganche entre torre y celda.
- 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.
- 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.
- El método `obtenerCelda`, devuelve la referencia a la celda del tablero.
- 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 retorna `null`.
- 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 retorna `null`.
- 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 retorna `null`.
- 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 la celda no pertenece al tablero se retorna "--".
- 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

3 Para su implementación es obligatorio utilizar un *array* de dos dimensiones de Java (con `[][]`) tal y como se ha visto en la sesión 3 de prácticas. **No se pueden utilizar estructuras de datos del paquete `java.util`.**

`[5][2]` se retorna `"a1c3"`). Si alguna celda no pertenece al tablero, su texto correspondiente es `"--"`.

- 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`.
- 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.

```
     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   -BN-     -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   -NM-     ----     ----     ----     ----     ----     ----     ----
    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 clase `Torre`:

- 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 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`[4].

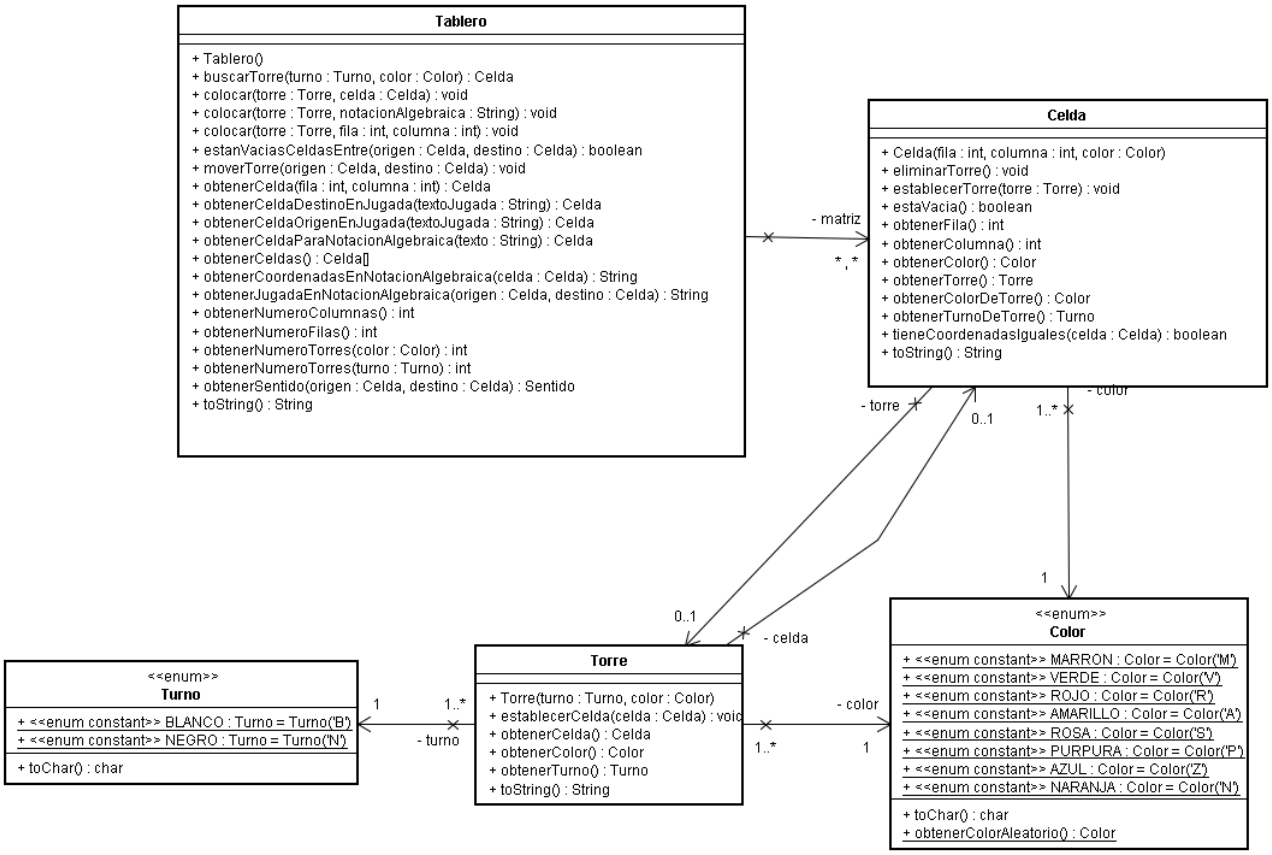


Ilustración 2: Modelo de clases del paquete juego.modelo

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 clase. Define la lógica de negocio o reglas del juego a implementar, comprobando la legalidad de las jugadas, realizando los movimientos si son legales, gestionando el cambio de turno y la actualización del los últimos colores de celda a los que ha movido cada jugador (ver Ilustración 3).

4 El símbolo ~ se traduce como acceso amigable.

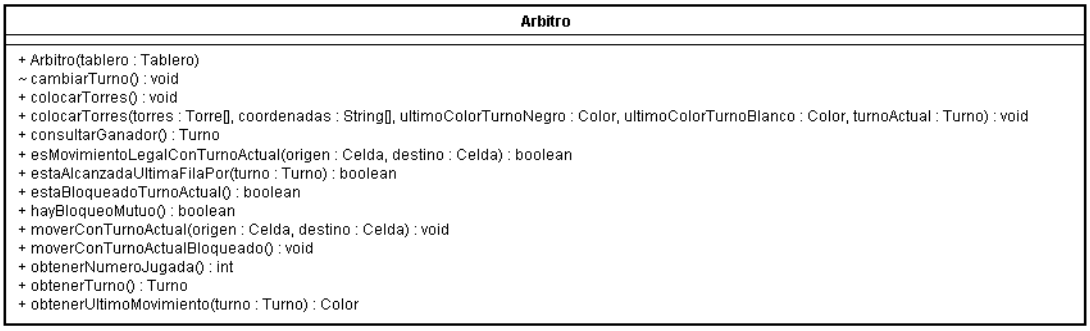


Ilustración 3: Diagrama de clases de `juego.control`

Comentarios respecto a la clase `Arbitro`:

- El constructor asigna el tablero con el que se va a jugar.
- El método `cambiarTurno` cambia el turno al otro jugador.
    - *Nota*: este método es amigable, y se limita su acceso a clases del mismo paquete.
- El método `colocarTorres()` inicializa el tablero asignado en el constructor, con todas las torres de ambos jugadores en sus filas correspondientes.
- El método `colocarTorres(Torre[], String[], Color, Color, Turno)` permite inicializar el tablero con una configuración diferente a la inicial, pasando un *array* de torres, un *array* de coordenadas en notacion algebraica donde colocar las torres, el color del último movimiento del jugador con turno negro, el color del último movimiento del jugador con turno blanco y el jugador con turno actual.
    - *Nota*: este método se implementa para ser utilizados en los tests automáticos y para facilitar al alumnado las pruebas y depuración del código. Se puede sustituir **temporalmente** la invocación al método `colocarTorres()` por este método para cargar partidas más simples de probar, sin tener que realizar tantos movimientos. Pero en la versión final entregada, no se debería utilizar.
- El método `consultarGanador` retorna el turno del ganador de la partida, bien por alcanzar la fila de salida del jugador contrario, o bien por existir bloqueo mutuo. Si no hay ganador devuelve `null`.
- El método `esMovimientoLegalConTurnoActual` dadas la celda origen y destino, devuelve `true` si es legal realizar el movimiento con el turno actual, o `false` en caso contrario.
- El método `estaAlcanzadaUltimaFilaPor` devuelve `true` si el jugador con el turno pasado ha alcanzado la fila de salida del jugador contrario, o `false` en caso contrario.
- El método `estaBloqueadoTurnoActual` devuelve `true` si el jugador con turno actual no puede mover la torre que corresponde, o `false` en caso contrario.
- El método `hayBloqueoMutuo` devuelve **`true`** si ninguno de los jugadores puede mover la torre que corresponde, **`false`** en caso contrario.
- El método `moverConTurnoActual` realiza el movimiento de la torre desde la celda origen a la celda destino. Se supone que previamente se ha comprobado la legalidad de la jugada y no es necesario volver a comprobarla. Debe ajustar el color de último movimiento para el turno actual y cambiar el turno, teniendo en cuenta que se ha finalizado una jugada.
- El método `moverConTurnoActualBloqueado` realizar un movimiento de "distancia cero" para el jugador con turno actual. Se supone que previamente se ha comprobado la situación de bloqueo del jugador y no es necesario volver a comprobarlo. Debe ajustar el color de último movimiento para el turno actual y cambiar el turno, teniendo en cuenta que se ha finalizado una jugada.

- El método `obtenerNumeroJugada` consulta el número de jugadas finalizadas en la partida. Todos los movimientos cuentan, tanto de torre a otra celda, y de "distancia cero".
- El método `obtenerTurno` consulta qué jugador tiene el turno actualmente.
- El método `obtenerUltimoMovimiento` devuelve el color de la celda donde realizó su último movimiento la torre del turno indicado.

Se omiten las relaciones de las clases del paquete control y modelo, deducibles a partir del enunciado y descripción de los métodos.

3.3 Paquete `juego.util`

Contiene una enumeración (ver en Ilustración 4).

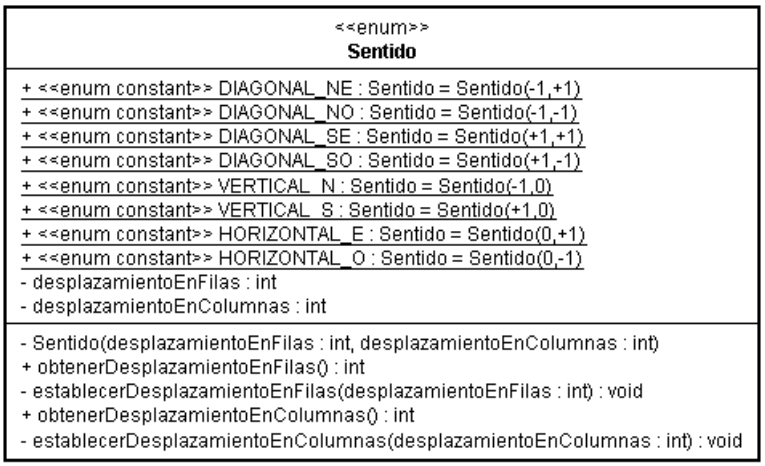


Ilustración 4: Diagrama de clases `juego.util`

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.

- **Este fichero se proporciona ya resuelto en UBUVirtual.**

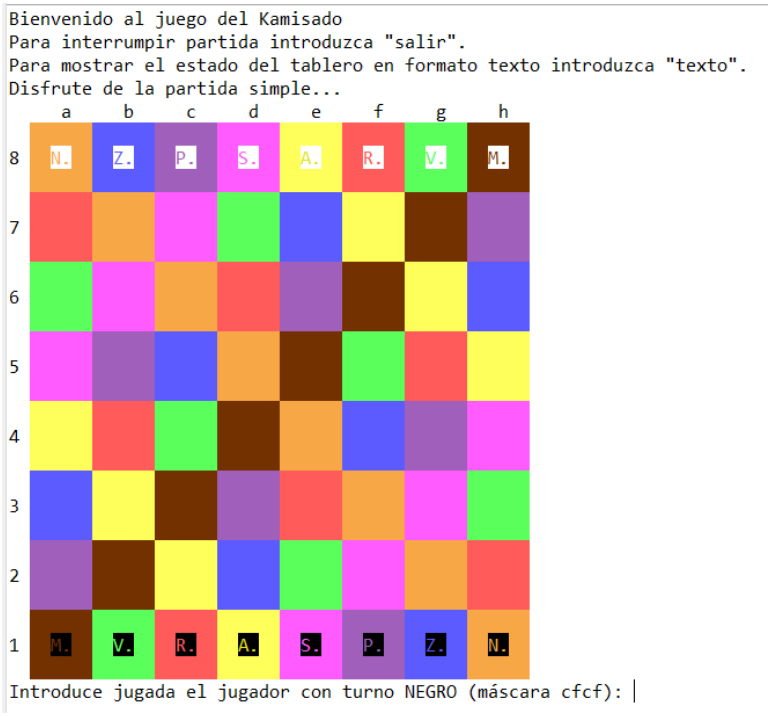
3.4 Paquete `juego.textui`

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

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

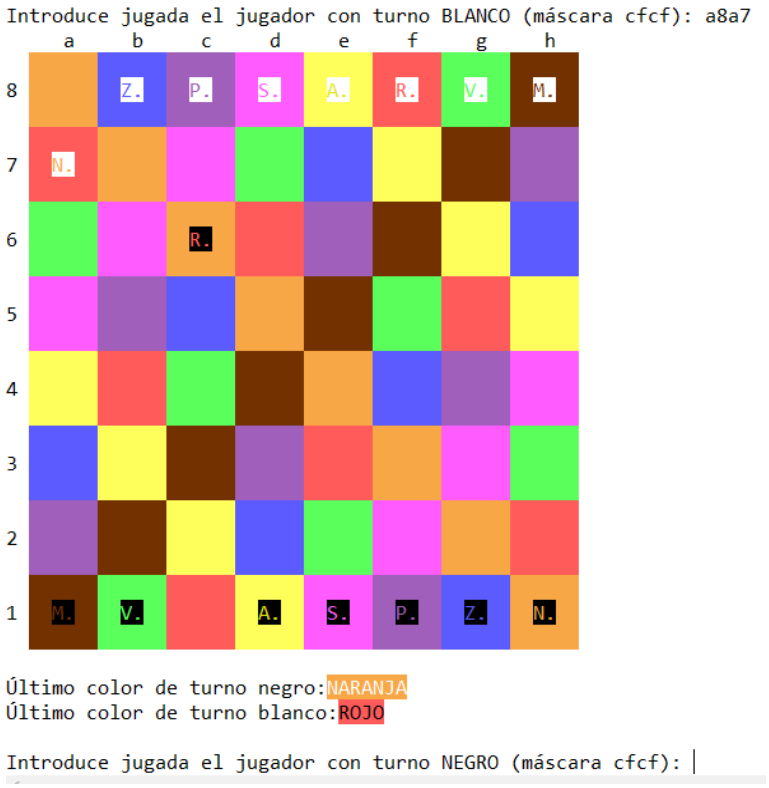
```
$> java juego.textui.Kamisado
```

La salida en pantalla debe ser similar a la siguiente (las capturas se realizan en Eclipse, en consola los colores de fondo y letras se alternarían a negro y blanco respectivamente):

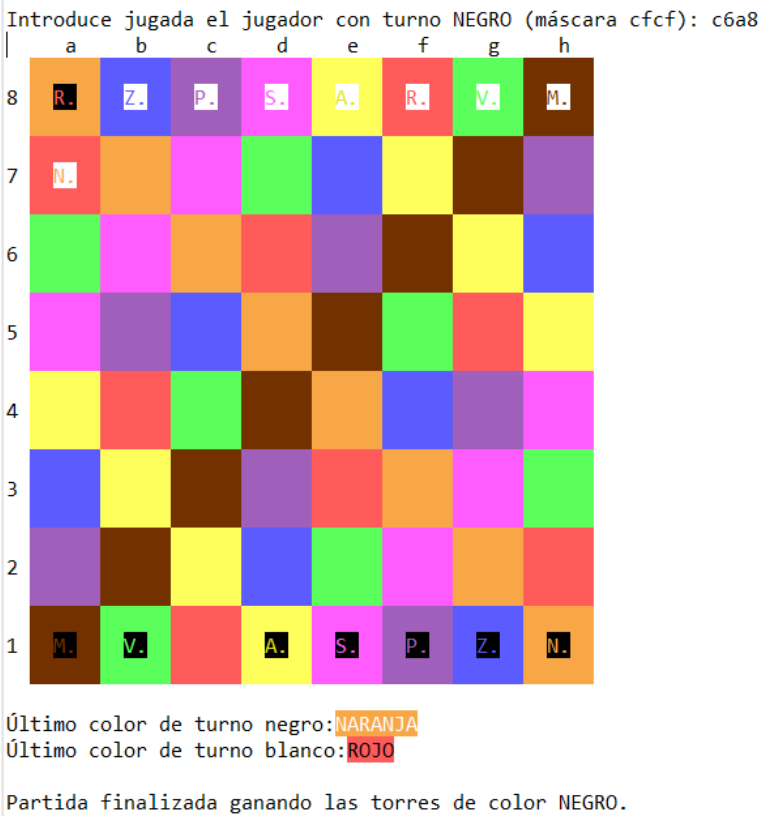


Suponiendo que en la primera jugada se juega de la celda `[7][2]` a `[2][2]` , torre de turno negro con color rojo a celda de color naranja en su vertical, introduciríamos la siguiente jugada `c1c6` y tendríamos la siguiente salida en pantalla:

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:



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 partida con victoria puesto que hemos alcanzado la fila del jugador contrario:

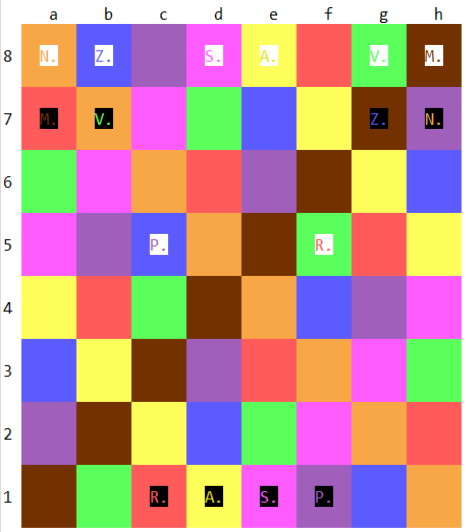


Si la jugada introducida **no es legal,** se debe informar del error al usuario, solicitando de nuevo que introduzca la jugada y sin saltar el turno. No hay límite en el número de reintentos.

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 muestra a continuación.

Si la jugada provoca un **bloqueo mutuo** se finaliza la partida informando del ganador, tal y como se muestra a continuación.

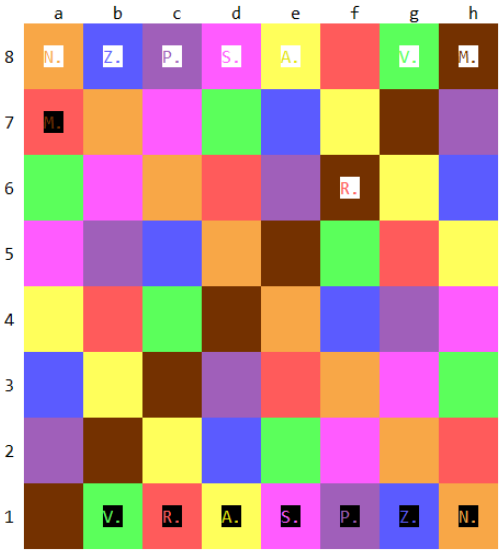
```
Introduce jugada el jugador con turno NEGRO (máscara cfcf): g1g7
La torre de turno BLANCO y color MARRON está bloqueada y pierde turno.
Se realiza un movimiento de distancia cero y se pierde el turno.
```




```
Último color de turno negro:MARRON
Último color de turno blanco:MARRON

Situacion de bloqueo mutuo.
Ganada la partida por el jugador con turno BLANCO,
porque no ha provocado el bloqueo.
```

```
Introduce jugada el jugador con turno BLANCO (máscara cfcf): f8f6
La torre de turno NEGRO y color MARRON está bloqueada y pierde turno.
Se realiza un movimiento de distancia cero y se pierde el turno.
```




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

Introduce jugada el jugador con turno BLANCO (máscara cfcf):
```

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`.

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

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

La interfaz inicial será similar a la mostrada en la Ilustración 5. En la parte superior se muestra el turno actual (e.g. NEGRO), y los últimos colores de cada turno (que inicialmente estarán sin definir).

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.

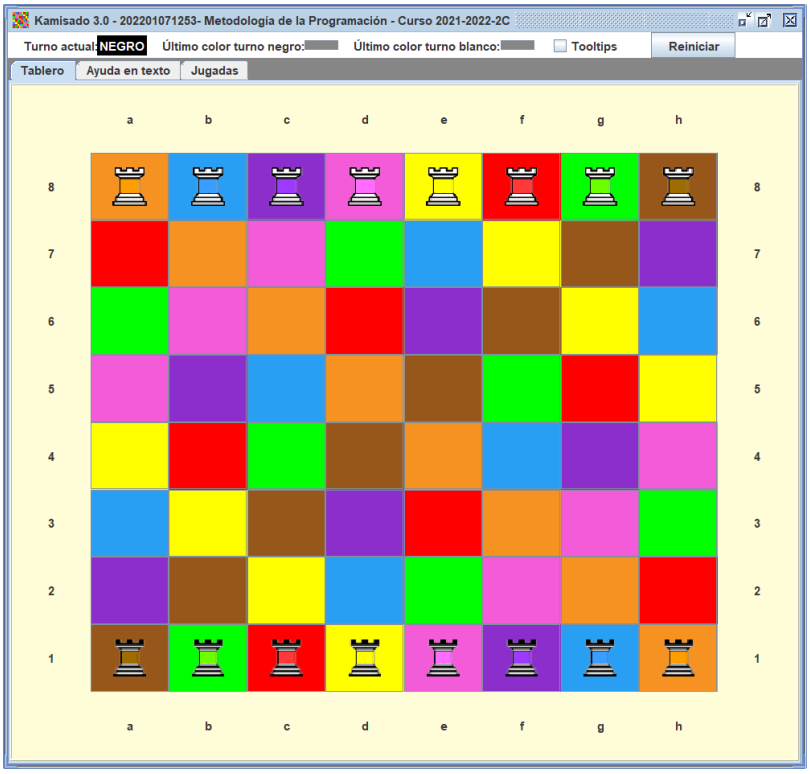


Ilustración 5: Pantalla de inicio en modo gráfico

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, 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.

Cuando se alcanza la fila contraria, la partida finaliza indicando el ganador, como se muestra en la Ilustración 6.

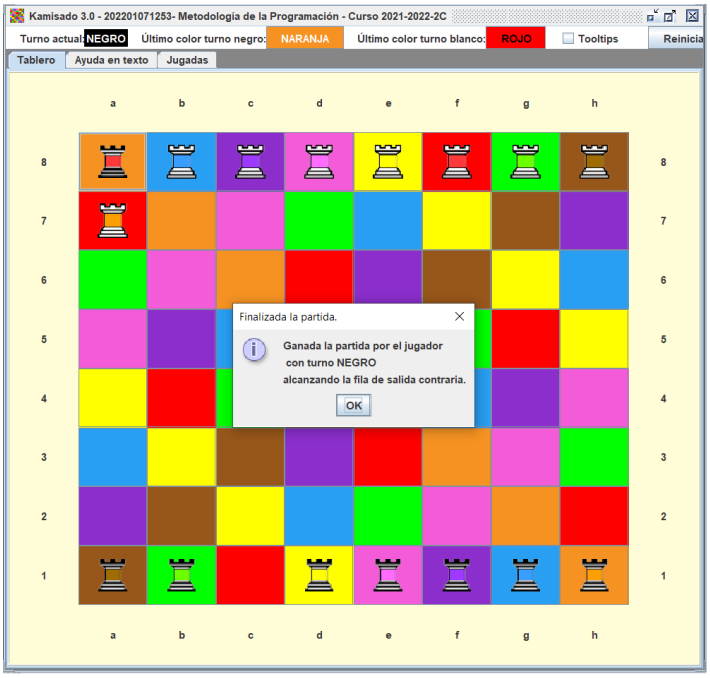


Ilustración 6: Partida finalizada

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 se muestra en la Ilustración 7.

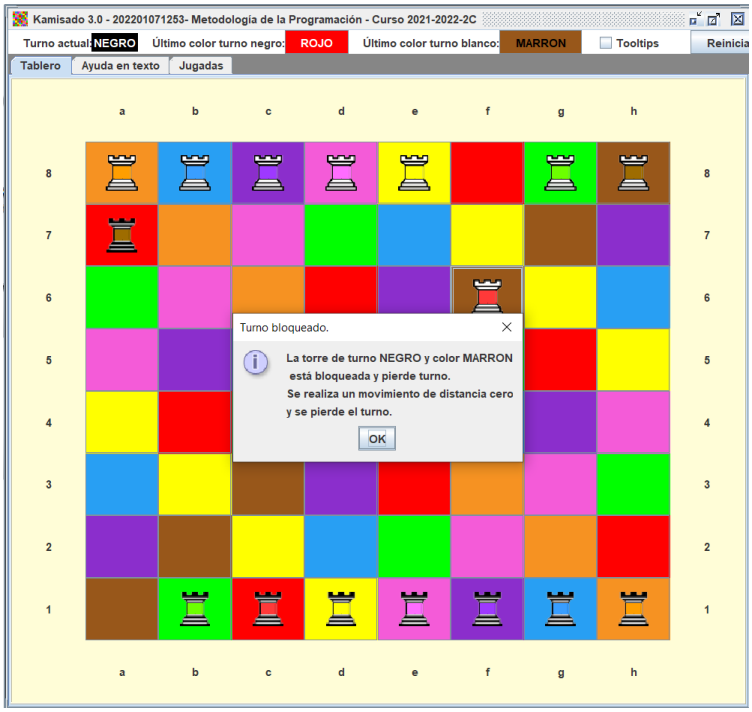


Ilustración 7: Bloqueo de turno con movimiento de "distancia cero"

Si se produce un bloqueo mutuo, en primer lugar se informará del bloqueo del jugador con siguiente turno (ver Ilustración 7), y a continuación de la situación de bloqueo mutuo, actualizando el estado de la partida e informando en pantalla de la finalización de partida y ganador, como se muestra en la Ilustración 8.

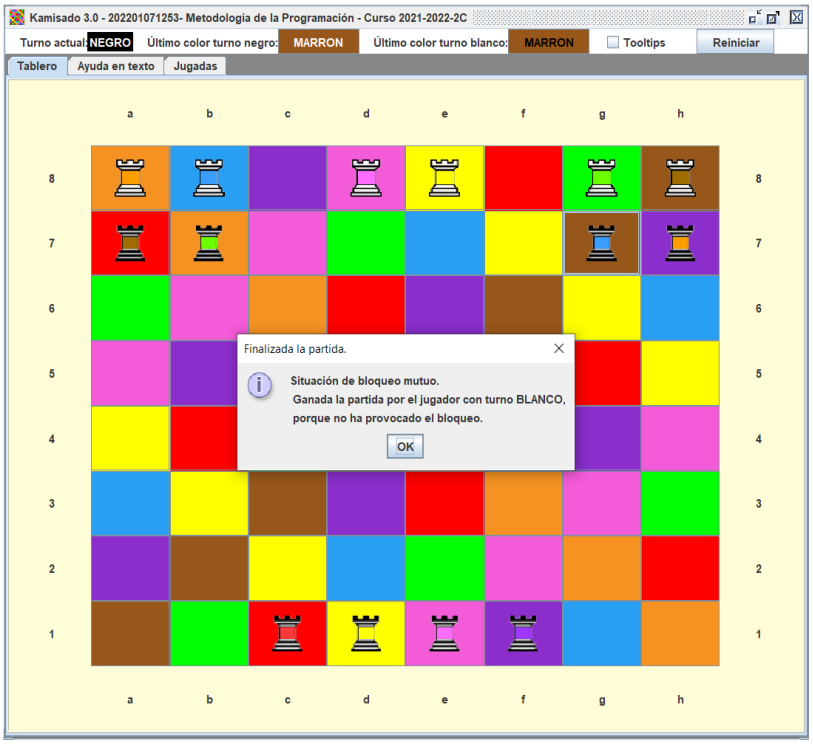


Ilustración 8: Bloqueo mutuo en partida

Las clases correspondientes a estos paquetes con la interfaz gráfica **se proporcionan en formato binario en un fichero con nombre** `kamisado-gui-lib-3.0.0.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 1 - EPO1-2C.**

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 1 – EPO1-2C.**
- 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-3.0.0.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.
- **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.

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

- 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[5] 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%)	20,00%
Kamisado (textui)	Corrección del algoritmo propuesto. Correcto funcionamiento.	10,00%
Arbitro	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.	25,00%
Tablero	Uso de modificadores de acceso. Atributos correctos. Constructor correcto. 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	25,00%
Celda Torre	Uso de modificadores de acceso. Atributos correctos. Cambios de estado correctos. Textos generados correctos. Métodos de cambio de estado.	15,00%
Color Turno	Correcta implementación de enumeraciones.	5,00%

Anexo. Visualización de colores en modo texto

Para resolver la visualización de colores en modo texto, se ha utilizado la biblioteca JColor (disponible en GitHub en https://github.com/dialex/JColor), a través del fichero `JColor-5.0.0.jar` que se proporciona por el profesorado. Dicha biblioteca utiliza los códigos de escape ANSI para mostrar colores en consola (ver https://es.wikipedia.org/wiki/C%C3%B3digo_escape_ANSI).

En el modo texto, en la clase `juego.textui.Kamisado`, se reutiliza dicha biblioteca para mostrar el tablero con colores[6].

Por defecto en sistemas operativos como GNU/Linux se mostrarán los colores correctamente en la consola del sistema operativo. Pero en la consola de Windows, esta opción puede estar desactivada (este *bug* está documentado en https://github.com/dialex/JColor/issues/54).

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

6 Debe estar el fichero Jcolor-5.0.0.jar en el CLASSPATH al ejecutar.

En caso de no mostrarse correctamente los caracteres con colores ANSI en Windows, se sugiere realizar el siguiente ajuste desde una consola **con permisos de administrador:**

```
$> reg add HKCU\Console /v VirtualTerminalLevel /t REG_DWORD /d 1
```

En el caso de Eclipse, en la vista de consola que viene por defecto, NO se mostrarán los colores, independientemente del sistema operativo utililzado. Para resolver este problema, se debe instalar un *plugin* adicional en el *Eclipse Marketplace* denominado *ANSI Escape in Console*. Dicho *plugin*, sustituye a la vista de consola por defecto de Eclipse, permitiendo mostrar los colores.

Para ello en el menú *Help*, en la opción *Eclipse Marketplace*, introducimos en el cuadro de búsqueda (*Search*) el texto "ANSI Escape" e instalamos el *plugin* (ver Ilustración 9).

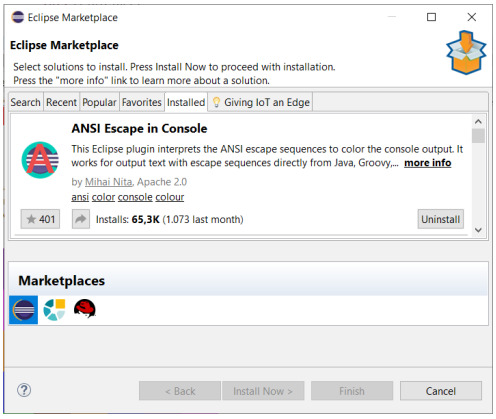


Ilustración 9: Plugin ANSI Escape in Console

Una vez instalado, se debe ajustar en el menú *Window* / *Preferences*, y elegimos *Ansi Console.* Seleccionamos la opción *Standard VGA colors* (ver Ilustración 10):

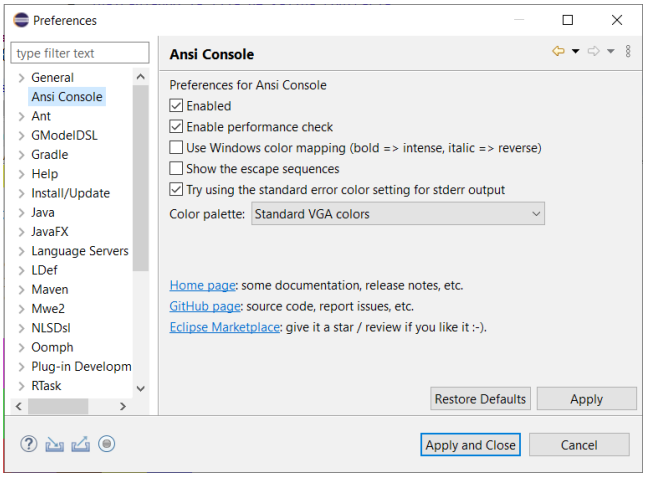


Ilustración 10: Configuración del plugin ANSI Console

En las siguientes ejecuciones en Eclipse puede saltar el siguiente aviso o *warning* (ver Ilustración 11), puesto que *ANSI Console* utiliza la configuración previa de *Console*. El problema viene por tener limitado el número de caracteres a mostrar en la consola.

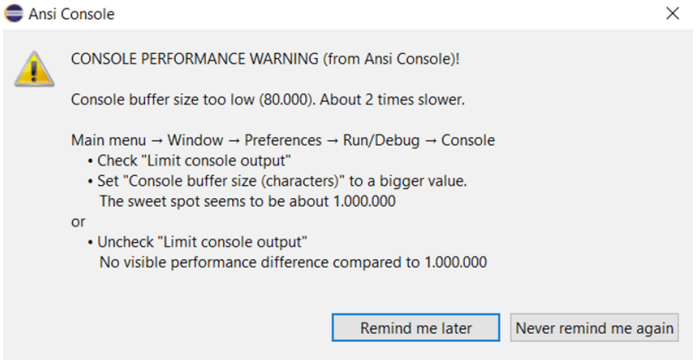


Ilustración 11: Warning de los problemas de rendimento de consola

Para evitar la aparición de dicho aviso, o bien seleccionamos el botón *"Never remind me again"* o bien seguimos las instrucciones mostradas, yendo al menú *Window / Preferences* y en la opción *Run/Debug → Console*, desmarcamos la opción *"Limit console output"* (ver Ilustración 12).

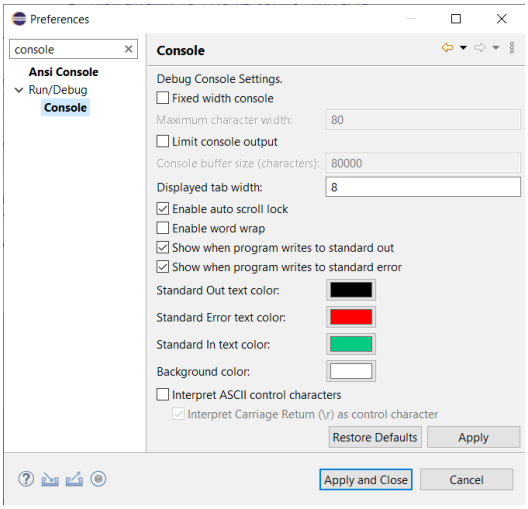


Ilustración 12: Configuración de límite de caracteres en consola

En la máquina virtual GNU/Linux proporcionada en la asignatura ya está instalado y configurado el *plugin,* por lo que no serían necesarios estos ajustes.

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