



Grado en Ingeniería Informática

METODOLOGÍA DE LA PROGRAMACIÓN

PRÁCTICA OBLIGATORIA 2 – PRIMERA CONVOCATORIA

NoventaGrados 2.0

Docentes:

Raúl Marticorena Sánchez
Ismael Ramos Pérez
José Miguel Ramírez Sanz



Índice de contenidos

1. INTRODUCCIÓN.....	3
2. OBJETIVOS.....	6
3. DESCRIPCIÓN.....	6
3.1 Paquete noventagrados.util.....	6
3.2 Paquete noventagrados.modelo.....	7
3.3 Paquete noventagrados.control.....	9
3.4 Paquete noventagrados.control.undo.....	12
3.5 Paquete noventagrados.textui.....	13
3.6 Paquete noventagrados.textui.excepcion.....	15
3.7 Paquetes noventagrados.gui y noventagrados.gui.images.....	15
4. ENTREGA DE LA PRÁCTICA.....	17
4.1 Fecha límite de entrega.....	17
4.2 Formato de entrega.....	17
4.3 Comentarios adicionales.....	19
4.4 Criterios de valoración.....	19
ANEXO 1. GENERACIÓN DE MÉTODOS EQUALS, HASHCODE Y TOSTRING EN ECLIPSE	21
ANEXO 2. EJECUCIÓN DE PRUEBAS AUTOMÁTICAS CON JUNIT.....	23
ANEXO 3. ORDEN DE IMPLEMENTACIÓN Y EJECUCIÓN ORDENADA DE SUITES.....	23
ANEXO 4. RECOMENDACIONES EN EL USO DE LA INTERFAZ LIST Y LA CLASE JAVA.UTIL.ARRAYLIST.....	24
RECURSOS.....	25



1. Introducción

El objetivo fundamental es implementar el juego de **Noventa Grados**, un juego de tablero abstracto de origen alemán.

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

- Inclusión de una **jerarquía de herencia para deshacer jugadas** (se recomienda revisar el Tema 4. Herencia, de teoría).
- **Inclusión y uso de interfaces y clases genéricas**, en la implementación de las clases, en sustitución del uso de *arrays*, así como la **creación de una clase genérica** (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 vs. no comprobables** (se recomienda revisar el Tema 6. Programación Defensiva y Tratamiento de Excepciones, de teoría).

A continuación se describen las **reglas a implementar**², ya tenidas en cuenta en la anterior versión.

Se utiliza un **tablero** de 7x7 celdas, para **dos contrincantes**: con siete piezas de color blanco y negro respectivamente. Una de esas siete piezas juega el papel de **reina**, considerando al resto **peones**.

Para la colocación inicial de las piezas se tomará siempre como referencia la siguiente figura en la Ilustración 1. Donde las piezas blancas ocupan la esquina superior izquierda, y las negras la esquina inferior derecha, con las correspondientes reinas en sus esquinas:

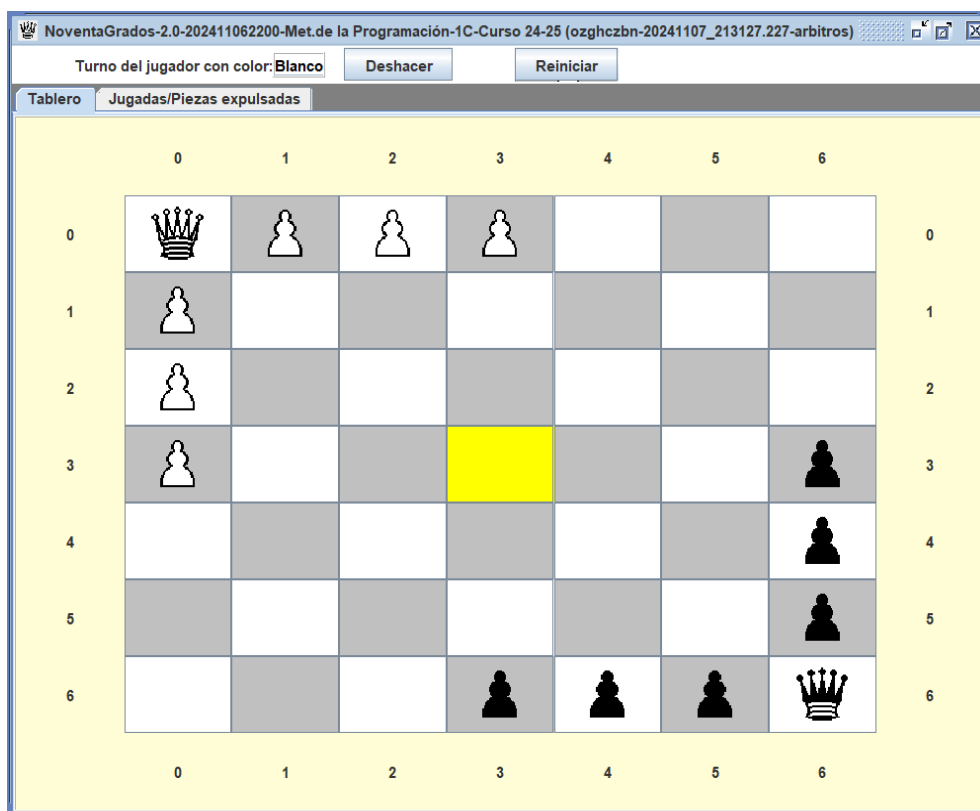


Ilustración 1: Ejemplo de inicio de partida

² En relación a las reglas, se toma como referencia siempre lo indicado en este enunciado y lo solicitado por los tests automáticos proporcionados.



Los objetivos para **ganar** la partida son iguales para ambos contrincantes: expulsar del tablero a la reina del contrincante o bien colocar su reina en el centro del tablero. Cumpliendo cualquiera de estas dos condiciones, se **gana** la partida.

En cada turno los **jugadores** podrán desplazar **solo una** de sus **piezas** dentro del tablero. Los movimientos solo pueden ser en **horizontal y vertical**. La única distancia válida a la que se puede desplazar **está determinada por el número de piezas que se encuentren en su perpendicular** con independencia del color y contándose a sí misma, de ahí el nombre del juego, puesto que siempre hay que contar en **90 grados** cuántas piezas hay.

Al desplazarse la pieza, el efecto que se produce en **las piezas con las que se choca, es que se desplazan en ese mismo sentido (literalmente, se empuja a las piezas)**. **Al empujar**, puede darse la situación que se **expulsa o echa a una pieza del tablero**, pasando a su correspondiente caja.

Se permite que un contrincante provoque su **propia derrota** expulsando a su reina del tablero. Si al empujar se expulsan ambas reinas se considerará una situación de **empate**.

No se consideran otras reglas y siempre comienza la partida el contrincante **atacante** con **turno blanco**.

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

Paquete	Módulos	Número	Descripción
noventagrados.control	Arbitro Caja TableroConsultor	3 clases	Gestión de las reglas, cajas para almacenar piezas expulsadas y consultas complejas sobre el tablero.
noventagrados.control.undo	MaquinaDelTiempoConArbitros MaquinaDelTiempoConJugadas MecanismoDeDeshacer MecanismoDeDeshacerAbstracto	1 interfaz 1 clase abstracta 2 clases	Jerarquía de herencia con dos implementaciones concretas diferentes para deshacer jugadas realizadas previamente.
noventagrados.modelo	Celda Jugada Pieza Tablero	3 clases 1 tipo registro (record)	Modelo fundamental.
noventagrados.textui	NoventaGrados	1 clase	Interfaz de usuario en modo texto. Se proporciona parcialmente el código a completar en UBUVirtual.
noventagrados.textui.excepcion	OpcionNoDisponibleException	1 excepción	Excepción comprobable.
noventagrados.util	Color Coordenada Sentido TipoPieza	3 enumeraciones (enum) 1 tipo registro (record)	Tipos enumerados y registros reutilizados en el resto de paquetes.
noventagrados.gui	<Sin determinar>	<Sin determinar>	Interfaz gráfica (se proporcionan los binarios en formato .jar)
noventagrados.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

El interfaz gráfico con ventanas se proporciona ya resuelto (filas en color verde en Tabla 1) con el fichero binario correspondiente (en formato .jar). Este fichero está disponible en la plataforma UBUVirtual para su descarga.



Es labor del alumnado implementar/completar los **ficheros fuente** `.java` necesarios para el correcto cierre y ensamblaje del sistema (filas en amarillo en Tabla 1), tanto en modo texto y gráfico, junto con el resto de productos indicados a continuación. **Se recuerda que NO se utilizarán en estas prácticas módulos Java (`module`), por lo tanto no se debería tener ningún fichero `module-info.java` en el proyecto.**

Para ello se deben seguir las indicaciones dadas y los diagramas disponibles, **respetando los diseños y firmas de los métodos**, con sus correspondientes modificadores de acceso, quedando **a decisión del alumnado la inclusión de atributos y/o métodos privados o protegidos** (`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.
 - Completando su ejecución en modo gráfico.
- Generar la documentación correspondiente al código fuente 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 como JUnit).
- Aportar los *scripts* correspondientes para realizar el proceso completo de compilación, documentación y ejecución (tanto en modo texto, como gráfico) desde una consola del sistema operativo.

3. Descripción

A continuación se desglosan los distintos diagramas y descripciones de módulos, a tener en cuenta de cara a la implementación de la práctica. Algunos métodos de consulta (Ej: `obtener...`, `consultar...`, `es...`, `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. Los métodos `equals`, `hashCode` y `toString` se generarán automáticamente desde Eclipse (ver solución en Anexo 1).

Por convención de nombres en esta práctica:

- los métodos `obtener...` devuelven una **referencia al objeto**.
- los métodos `consultar...` devolverán una **referencia a un clon profundo** del objeto consultado, con la **excepción** de que devuelva un **tipo enumerado/registro** o **tipo primitivo**, devolviendo en este caso solo la referencia a dicho valor **inmutable**, o bien el valor primitivo, respectivamente.

La implementación de los métodos `clonar` que solicitan una **clonación**, se resolverá utilizando los constructores de las clases correspondientes, instanciando nuevos objetos y pasando como argumentos los nuevos valores que sean necesarios, de tal forma que se clone en profundidad, y no superficialmente. **NO se redefinirá el método `clone` (heredado de `java.lang.Object`), ni es necesario implementar interfaces como `Cloneable`.**

Algunos de los métodos solicitados solo se utilizarán en los *tests* automáticos para comprobar la corrección de la implementación dada.

Se recomienda consultar los **diagramas de clases** correspondientes, donde se muestran los conjuntos completos de métodos públicos a implementar en cada clase, con sus signaturas completas en notación UML.

3.1 Paquete `noventagrados.util`

Contiene tres enumeraciones y un tipo registro (ver en Ilustración 2).



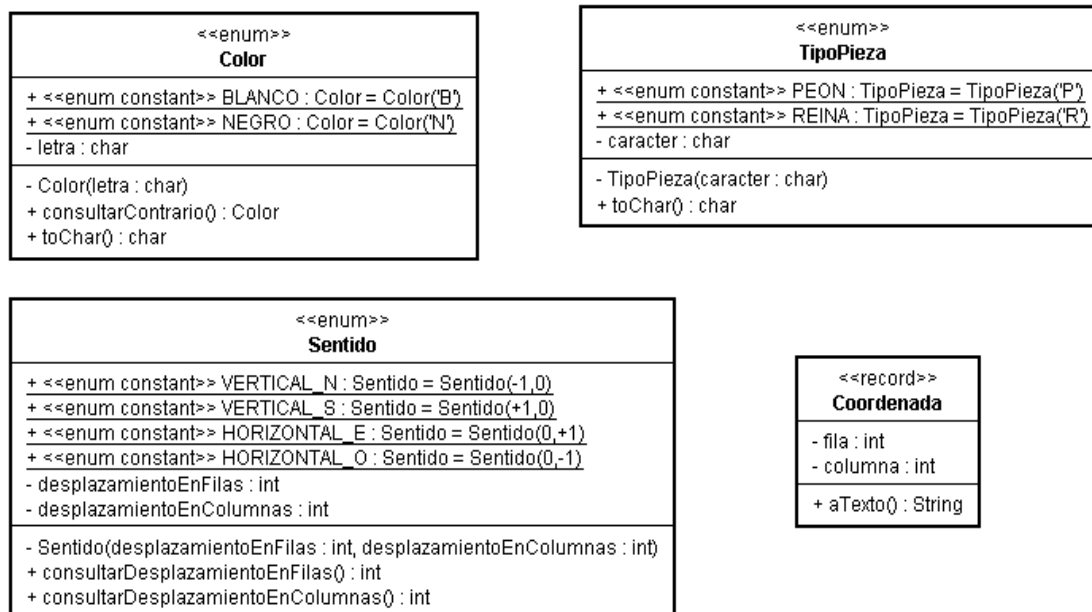


Ilustración 2: Diagrama de clases del `noventagradados.util`

- La enumeración `Color` define los dos colores del turno, asignando el carácter 'B' para el color blanco y 'N' para el color negro. Permite consultar el color contrario al actual, con el método `consultarContrario`.
- El tipo registro `Coordinada` almacena el valor de fila y columna relativo a la posición de la celda en un tablero. Adicionalmente, proporciona un método `aTexto` que recupera la fila y columna en formato cadena de caracteres (ej. "00", "23", "64", etc.)
- La enumeración `Sentido` modela los cuatro sentidos en los que se pueden realizar los movimientos: `VERTICAL_N`, `VERTICAL_S`, `HORIZONTAL_E`, `HORIZONTAL_O`. Cada valor del tipo enumerado lleva ligado tanto el valor de desplazamiento en filas y columnas. Se añaden los correspondientes métodos de consulta para ambos valores.
- La enumeración `TipoPieza` define los dos tipos de piezas posibles: `PEON` y `REINA`, asociando a cada uno su correspondiente carácter ('P' y 'R' respectivamente) y con su método de consulta `toChar`.

3.2 Paquete `noventagradados.modelo`

El paquete está formado por tres clases y un tipo registro. A continuación se muestra el diagrama de clases completo para el paquete `noventagradados.modelo`³.

3 El símbolo ~ se traduce como acceso amigable.



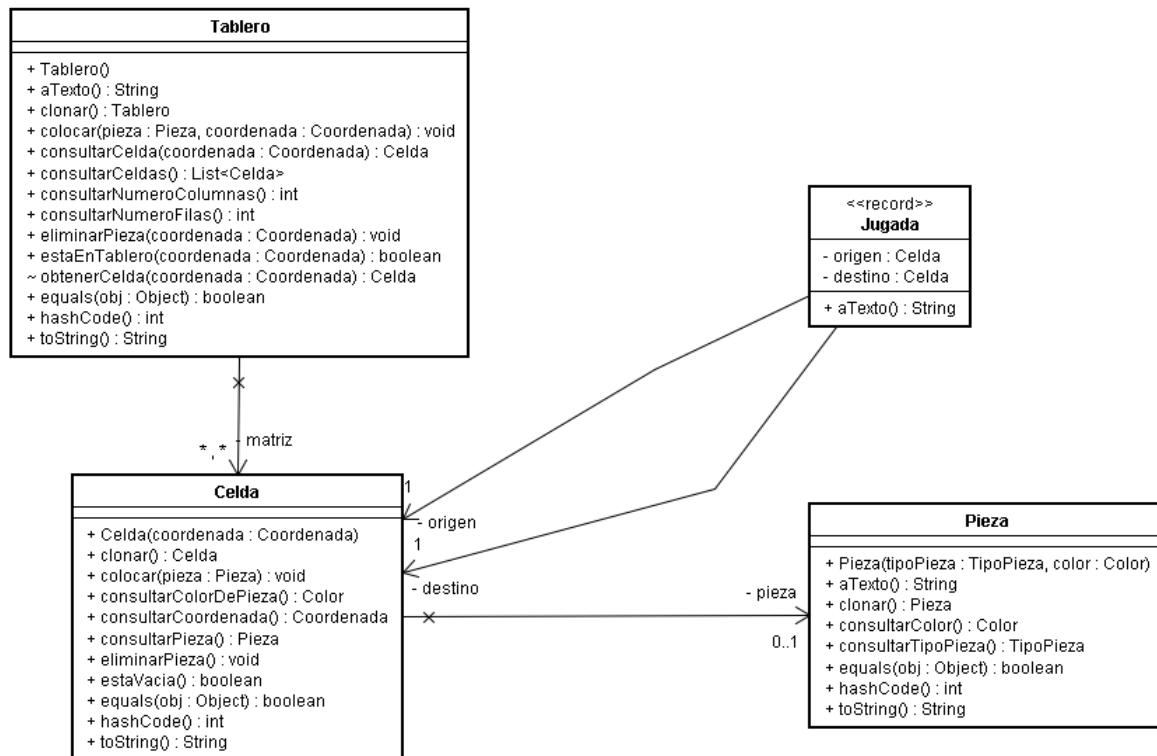


Ilustración 3: Diagrama de clases del paquete *noventagrados.modelo*.

Comentarios respecto a la clase *Celda*:

- Incluye un constructor con la coordenada asociada.
- El método `clonar` devuelve un **clon en profundidad** de la celda actual.
- El método `colocar` sitúa o relaciona una pieza en la celda actual.
- El método `consultarColorDePieza` retorna el color de la pieza actual, si no está vacía, o nulo en caso contrario.
- El método `eliminarPieza` elimina la asignación actual de pieza en la celda, dejando en su lugar un valor nulo.
- El método `estaVacia` consulta si hay o no una pieza colocada en dicha celda.

Comentarios respecto al tipo registro *Jugada*:

- Se inicializa con la coordenada origen y destino de la jugada.
- El método `aTexto` devuelve el texto asociado al par de coordenadas correspondientes a las celdas y origen separados por un guion (e.g. "00-11", "23-45", etc.)

Comentarios respecto a la clase *Pieza*:

- Se inicializa con el tipo de la pieza y el color.
- El método `aTexto` devuelve en formato cadena de caracteres el texto correspondiente al tipo de pieza y color (e.g. "PB" para un peón blanco, "RN" para una reina negra).
- El método `clonar` devuelve un **clon en profundidad** de la pieza actual.

Comentarios respecto a la clase *Tablero*:



- Un tablero se considera como un conjunto de celdas, cada una en una coordenada (e.g. (fila,columna)).
 - Suponiendo que el tablero es de 7 filas x 7 columnas, entonces se tendrá: (0, 0) las coordenadas de la esquina superior izquierda, (0, 6) las coordenadas de la esquina superior derecha, (6, 0) las coordenadas de la esquina inferior izquierda y (6, 6) las coordenadas de la esquina inferior derecha. Se numera de izquierda a derecha y en sentido descendente de arriba a abajo.
- 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 un tipo genérico se puede definir en función de sí mismo como se ha visto en el Tema 5 (Ej: `Pila<Pila<Circulo>>`). Por lo tanto se debería utilizar una lista de listas (ver Anexo 4).
- El método `aTexto` devuelve el estado del tablero con las piezas actualmente colocadas en formato cadena de caracteres, para mostrar en pantalla. El texto generado se debe corresponder con la salida esperada en la ejecución en modo texto (ver ejemplos en la Sec. 3.5) y en los *tests*.
- El método `clonar` devuelve un **clon en profundidad** del tablero actual.
- El método `colocar` coloca en la coordenada indicada la pieza pasada como argumento. Si las coordenadas no están en el tablero o la pieza valiera nulo, no se hace nada.
- El método `consultarCelda` devuelve un **clon en profundidad** de la celda con las coordenadas indicadas. Si las coordenadas no están en el tablero devuelve un valor `null`.
- El método `consultarCeldas` devuelve una **lista** de celdas, con **clones en profundidad** de todas las celdas del tablero, recorriendo las celdas de arriba hacia abajo, y de izquierda a derecha.
- El método `eliminarPieza` elimina la pieza en la celda con la coordenada indicada. Si la coordenada no está en el tablero no se realiza ninguna acción.
- El método `obtenerCelda` devuelve la referencia a la celda con la coordenada indicada. Si la coordenada no está en el tablero, devuelve un valor nulo.

3.3 Paquete `noventagrados.control`

El paquete contiene tres clases. Define parte de la lógica de negocio y coordinación del resto de objetos, comprobando la legalidad de las jugadas, realizando los movimientos si son legales, completando las expulsiones del tablero, gestionando el cambio de turno y la comprobación de situaciones de finalización de la partida (ver Ilustración 4).

Incluye la implementación de las cajas donde se dejan las piezas expulsadas de tablero y una clase de utilidad para consultas complejas sobre el estado de un tablero actual.

En todas estas clases, se debe **sustituir** el uso de *arrays* por el uso de `java.util.List` y `java.util.ArrayList` con genericidad.



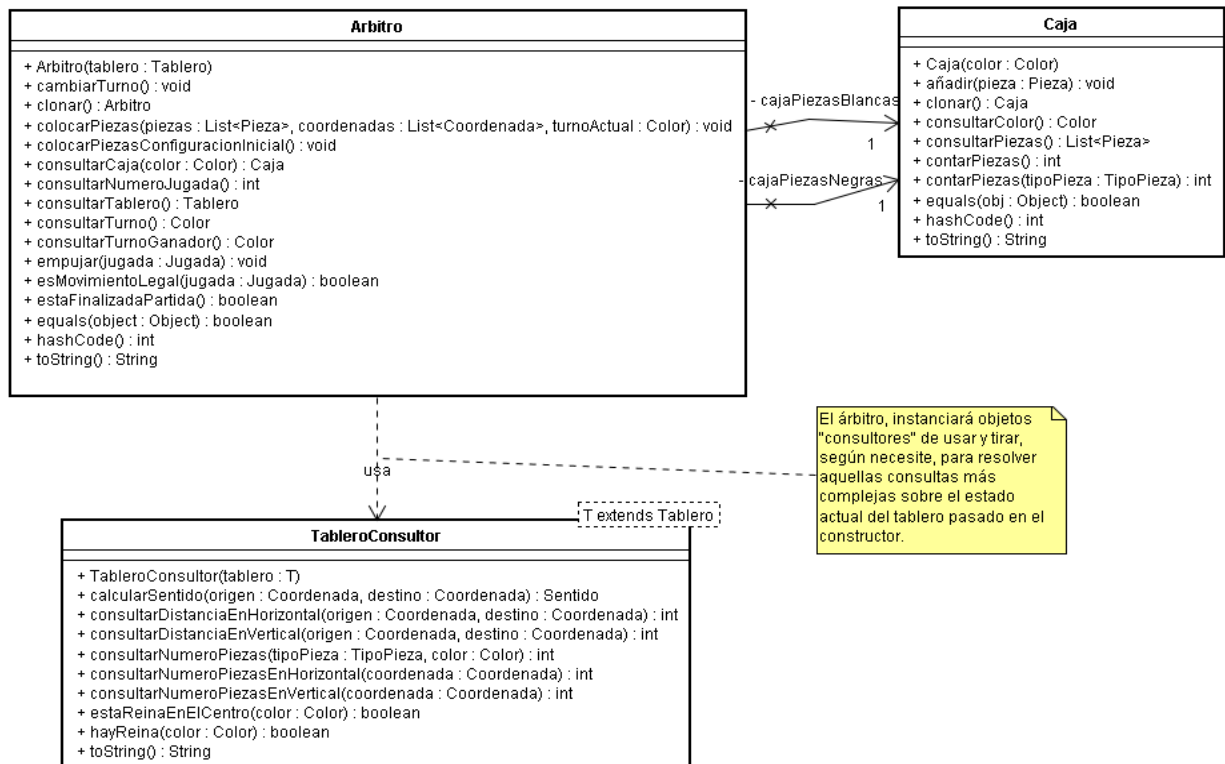


Ilustración 4: Diagrama de clases del paquete `noventagrados.control`.

Comentarios respecto a la clase `Arbitro`:

- El constructor inicializa la partida con un tablero y con su contador de jugadas a cero. Inicialmente no hay turno asignado, hasta que no se coloquen las piezas posteriormente.
- El método `cambiarTurno` cambia el turno al otro contrincante.
- El método `clonar` retorna un clon en profundidad del árbitro actual.
- El método `colocarPiezas` coloca piezas correspondientes a una lista de tipos de pieza, con sus coordenadas correspondientes, e inicializando el turno actual al color indicado. Su uso está **destinado a los tests automáticos** y a las **pruebas adicionales** que se quieran hacer (e.g. en un método `main` en la propia clase), inicializando una partida con las piezas colocadas discrecionalmente, al igual que el turno.
- El método `colocarPiezasConfiguracionInicial` coloca las piezas correspondientes a la configuración de inicio del juego con sus 14 piezas (ver Ilustración 1), e inicializando siempre el turno para el atacante con piezas blancas.
- El método `consultarCaja` devuelve la caja conteniendo las piezas expulsadas del color indicado.
- El método `consultarNumeroJugada` devuelve el número de jugadas realizadas hasta el momento.
- El método `consultarTablero` devuelve un **clon en profundidad** del tablero actual.
- El método `consultarTurno` devuelve el turno actual, que puede realizar la siguiente jugada.
- El método `consultarTurnoGanador` devuelve el turno del ganador actual o `null` si no hay ganador.
- El método `empujar` desplaza la pieza en origen hasta destino empujando las piezas que se encuentre en su sentido de desplazamiento, incrementando el número de jugada. Si las piezas son expulsadas del tablero, deben añadirse a la caja correspondiente. Se asume que ya se ha comprobado previamente que la jugada es legal, no siendo necesario volverlo a comprobar.



- El método `esMovimientoLegal` comprueba la legalidad de la jugada, según las reglas del juego. Debe aplicar solo las reglas descritas en la Sec. 1, incluyendo la comprobación de si la partida ha finalizado previamente, en cuyo caso no sería legal mover.
- El método `estaFinalizadaPartida` comprueba si se da cualquiera de las condiciones de finalización actualmente.

Comentarios respecto a la clase `Caja`:

- El constructor inicializa la caja vacía con su color asignado.
- El método `añadirPieza` asigna la pieza a dicha caja.
- El método `consultarPiezas` devuelve una lista de una dimensión, con **clones en profundidad** de todas las piezas en la caja.
- El método `contarPiezas()` devuelve el número total de piezas contenidas.
- El método `contarPiezas(TipoPieza)` devuelve el número de piezas contenidas de un determinado tipo.

Comentarios respecto a la clase `TableroConsultor`:

- Se trata ahora de una clase genérica, parametrizable solo por subtipos de tipo `Tablero`.
- El constructor se inicializa con un tipo restringido por tablero, sobre el que se realizarán solo consultas (se utilizará desde el `Arbitro`, siempre pasando un clon para evitar efectos laterales).
- El método `calcularSentido` retorna el sentido entre coordenadas, o bien `null` si no es un sentido válido en este juego.
- El método `consultarDistanciaEnHorizontal` retorna la distancia entre dos coordenadas que estén en la misma horizontal (e.g. distancia de (0,0) a (0,2) sería 2, distancia de (1,4) a (1,1) sería 3). Si no están en la misma horizontal retorna -1.
- El método `consultarDistanciaEnVertical` retorna la distancia entre dos coordenadas que estén en la misma vertical (e.g. distancia de (0,0) a (2,0) sería 2, distancia de (4,4) a (1,4) sería 3). Si no están en la misma vertical retorna -1.
- El método `consultarNumeroPiezas` retorna el número de piezas del tipo y color indicado sobre el tablero.
- El método `consultarNumeroPiezasEnHorizontal` retorna el número de piezas contenidas en la misma horizontal de la coordenada dada, incluyendo a la pieza en dicha coordenada, si existe, y con independencia del color de las piezas.
- El método `consultarNumeroPiezasEnVertical` retorna el número de piezas contenidas en la misma vertical de la coordenada dada, incluyendo a la pieza en dicha coordenada, si existe, y con independencia del color de las piezas.
- El método `estaReinaEnElCentro` consulta si la reina del color indicado ocupa la celda central.
- El método `hayReina` comprueba si la reina del color indicado está todavía sobre el tablero.

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



3.4 Paquete `noventagrados.control.undo`

En este paquete se implementa el **mecanismo para deshacer jugadas** ("undo" en inglés). Se compone de una jerarquía de herencia, con una `interface`, una clase abstracta y dos clases descendientes concretas. Se implementan dos formas distintas de deshacer: almacenando las jugadas vs. almacenando clones de los árbitros.

Comentarios respecto a la interfaz `MecanismoDeDeshacer`:

- El método `consultarArbitroActual` devuelve un clon en profundidad del árbitro en el estado actual, según se hayan hecho y deshecho las jugadas. Si no hay jugadas en el histórico, el árbitro estará en su estado inicial, con las piezas colocadas sobre el tablero y las cajas vacías.
- El método `consultarNumeroJugadasEnHistorico` devuelve el número de jugadas que pueden deshacerse hasta el momento. Al hacer jugadas, se incrementará dicho número y al deshacer se decrementará.
- El método `deshacerJugada` deshace la última jugada realizada.
- El método `hacerJugada` recibe la última jugada realizada para guardar sus efectos.
- El método `obtenerFechaInicio` devuelve la fecha en la que se inicializa el mecanismo de deshacer.

Comentarios respecto a la clase abstracta `MecanismoDeDeshacerAbstracto`:

- Clase abstracta donde se deben declarar atributos y métodos comunes que serán necesarios en los descendientes implementando la interfaz `MecanismoDeDeshacer`.
- Define el constructor de la máquina del tiempo, inicializándose con la fecha actual (una instancia de `java.util.Date`, usando el constructor por defecto).

Comentarios respecto a la clase `MaquinaDelTiempoConJugadas`:

- Su implementación interna se basa en almacenar el histórico de las jugadas realizadas (objetos de tipo `record Jugada`) en una lista genérica, según se van realizando.
- Cuando se pide `consultarArbitroActual`, se debe instanciar un nuevo árbitro (en su estado inicial con las piezas colocadas) y se aplican las jugadas grabadas previamente, teniendo en cuenta los cambios de turno, devolviendo dicho árbitro.
- El resto de detalles de implementación de la clases son discrecionales y responsabilidad del alumnado.

Comentarios respecto a la clase `MaquinaDelTiempoConArbitros`:

- Su implementación interna se basa en almacenar el histórico de clones de los árbitros (objetos de tipo `Arbitro`) en una lista genérica, según se van haciendo jugadas.
- Cuando se pide `consultarArbitroActual`, devuelve el último clon de la partida almacenado en la lista, según se hayan hecho y deshecho jugadas.
- El resto de detalles de implementación de la clases son discrecionales y responsabilidad del alumnado.



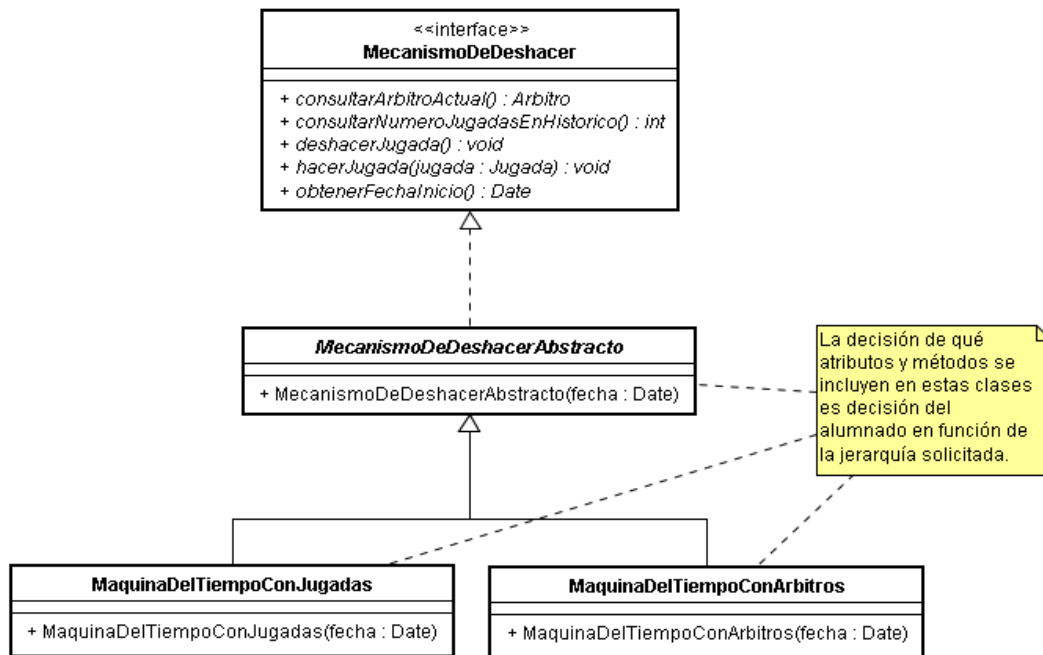


Ilustración 5: Modelo de clases de paquete noventagradados.control.undo

Se omiten las relaciones con el resto de paquetes, deducibles a partir del enunciado y descripción de los métodos.

3.5 Paquete noventagradados.textui

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

Debe incluir en la implementación del método `main` el **tratamiento de excepciones** adecuado para tratar tanto las **posibles excepciones comprobables y no comprobables**, dando los mensajes de error adecuados (revisar los métodos ya resueltos en dicha clase). **Se debe garantizar que la ejecución en modo texto es robusta, dando una respuesta amigable ante cualquier error, ya sea por excepción comprobable vs. no comprobable, que se produzca.**

El método `extraerModoDeshacer` deber completarse, analizando el *array* de argumentos, comprobando que si contiene una primera cadena de texto, esta tiene que ser "jugadas" o "arbitros", inicializando con dicho valor la variable `configuracion`. Si la primera cadena pasada no es ninguna de estas dos, se lanzará una excepción **comprobable** `OpcionNoDisponibleException`. Si el *array* no contiene elementos, se toma como valor por defecto "jugadas" para la variable `configuracion`.

También debe tenerse en cuenta las excepciones no comprobables lanzadas por el método `seleccionarMecanismoDeshacer`, ya resuelto.

En función del argumento pasado, se instanciará la máquina del tiempo concreta correspondiente.

Un ejemplo de sintaxis de invocación (sin detallar cómo debe configurarse el `classpath`) activando la opción de deshacer con *jugadas* sería::

```
$> java noventagradados.textui.NoventaGrados jugadas
```

La salida en pantalla debe ser similar a la siguiente:



Bienvenido al juego de Noventa Grados 2.0 - Máquina del tiempo con jugadas
 Introduzca sus jugadas con el formato dd-dd donde d es un dígito en el rango [0, 6] (por ejemplo 00-04 o 65-63).
 Para interrumpir la partida introduzca "salir".
 Para deshacer la última jugada introduzca "deshacer".
 Disfrute de la partida...

```

0 RB PB PB PB -- -- --
1 PB -- -- -- -- -- --
2 PB -- -- -- -- -- --
3 PB -- -- -- -- -- PN
4 -- -- -- -- -- -- PN
5 -- -- -- -- -- -- PN
6 -- -- -- PN PN PN RN
  0  1  2  3  4  5  6

```

Introduce jugada turno con piezas de color BLANCO:

Suponiendo que en la primera jugada se quiere mover blancas de (0, 3) a (4, 3), se introduce como entrada 03-43, mostrando a continuación el estado de la partida en pantalla:

Introduce jugada turno con piezas de color BLANCO: 03-43

```

0 RB PB PB -- -- -- --
1 PB -- -- -- -- -- --
2 PB -- -- -- -- -- --
3 PB -- -- -- -- -- PN
4 -- -- -- PB -- -- PN
5 -- -- -- -- -- -- PN
6 -- -- -- PN PN PN RN
  0  1  2  3  4  5  6

```

Introduce jugada turno con piezas de color NEGRO:

Se siguen introduciendo jugadas, hasta que se alcance la finalización de partida, dando un mensaje en pantalla como el siguiente en función del ganador:

Ha ganado la partida el turno con piezas de color NEGRO.
 Partida finalizada.

O bien en el caso de que ambas reinas sean empujadas a la vez fuera del tablero:

Empate con ambas reinas empujadas fuera del tablero.
 Partida finalizada.

Si la jugada introducida **no es legal**, o el **formato no es correcto**, 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 erróneos.

Si en el algún momento el usuario introduce `deshacer` se deshace la última jugada realizada y se restaura la partida a su estado previo, mostrando el estado del tablero en pantalla y pidiendo una nueva jugada.

Si en algún momento el usuario introduce `salir` se interrumpe la partida y se finaliza simplemente mostrando en pantalla:

Partida finalizada.

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



el “algoritmo” principal. Se pueden añadir métodos privados adicionales, pero no amigables o públicos.

3.6 Paquete `noventagrados.textui.excepcion`

Contiene `OpcionNoDisponibleException` que es una **excepción comprobable**, que obligatoriamente debe heredar de `java.lang.Exception`. Incorpora los cuatro constructores habituales, mostrados en la Ilustración 7 junto con su valor constante `serialVersionUID`⁴.

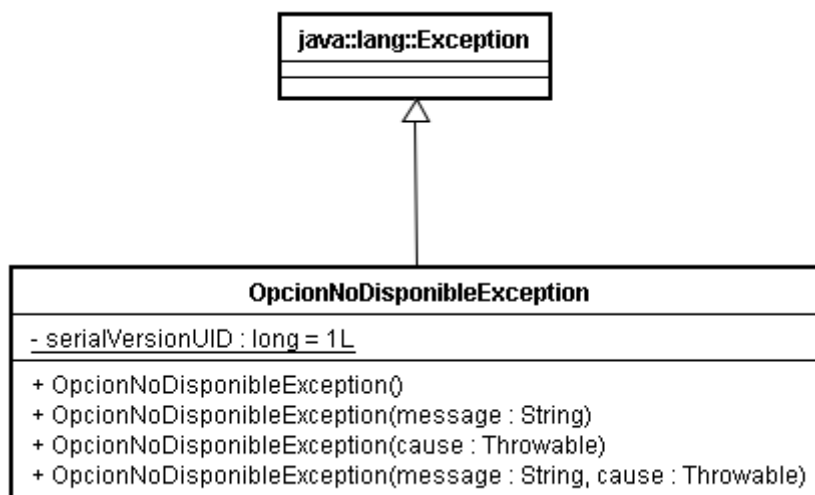


Ilustración 6: Excepción comprobable a implementar

3.7 Paquetes `noventagrados.gui` y `noventagrados.gui.images`

Estos paquetes implementan la **interfaz gráfica del juego**. Se proporciona **ya resuelta** por el profesorado en un fichero `.jar` (en formato binario ya compilado).

La clase raíz del sistema es `noventagrados.gui.NoventaGrados`. Se le puede pasar como argumento adicional `jugadas` o `arbitros` para utilizar cada uno de las máquinas del tiempo implementadas. Si no se introduce ningún valor, se toma por defecto el valor `arbitros`.

Ejemplo de invocación (sin detallar cómo debe configurarse el `classpath` quedando como ejercicio para el alumnado añadiendo el fichero `.jar` proporcionado) :

```
$> java noventagrados.gui.NoventaGrados arbitros
```

La interfaz inicial al ejecutar será la mostrada en la Ilustración 7. En la parte superior se muestra el turno actual (e.g. BLANCO), un botón para deshacer la última jugada y un botón para reiniciar la partida.

⁴ Dicho valor se utiliza para la **serialización** de objetos (para su persistencia o transporte por la red habitualmente). En esta asignatura no tiene mayor utilidad y se añade solo para evitar *warnings* del entorno de desarrollo.



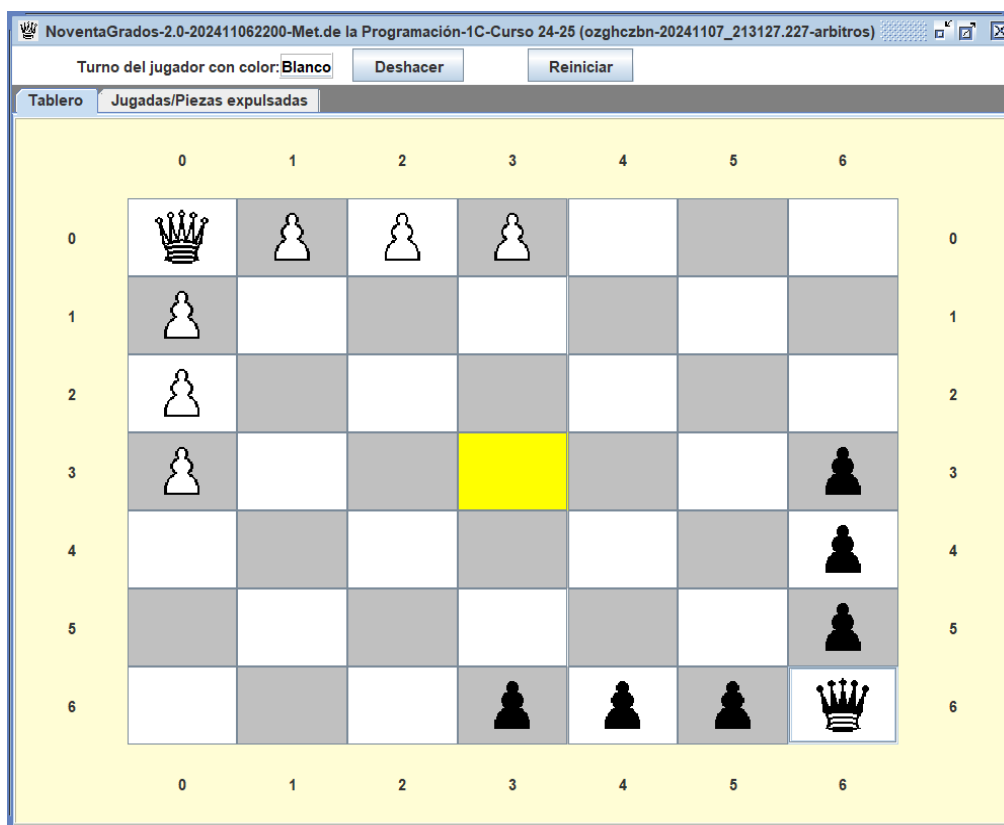


Ilustración 7: Pantalla de inicio en modo gráfico

Se proporciona una pestaña adicional:

- **Jugadas/Piezas expulsadas:** donde se visualizan la lista de jugadas y las piezas expulsadas en sus correspondientes cajas.

Para realizar una jugada, el jugador con turno actual debe seleccionar con el ratón sobre la pieza que quiere mover (origen), y luego seleccionar la celda del tablero donde quiere colocarla (destino). Si el movimiento es ilegal se mostrará un diálogo con el error en pantalla. Si es legal, se desplazará la pieza en el tablero, empujando a las piezas que haya en su trayectoria. En caso de expulsar a una pieza del tablero, esta desaparecerá del mismo, y se mostrará en su caja correspondiente. Si la partida no ha finalizado se cambia el turno.

En cualquier momento durante el juego se puede presionar el botón *Deshacer*, para **restaurar el árbitro a un estado previo, deshaciendo la última jugada**. Se puede presionar dicho botón tantas veces seguidas como se quiera, hasta volver al estado inicial de la partida con el tablero con las piezas iniciales y las cajas vacías.

Cuando se finaliza la partida se muestra un diálogo informando del turno ganador (ver en la Ilustración 8 un ejemplo de victoria de blancas llegando con reina al centro). Cerrando dicho diálogo (presionando *Ok*) se reinicia la partida.



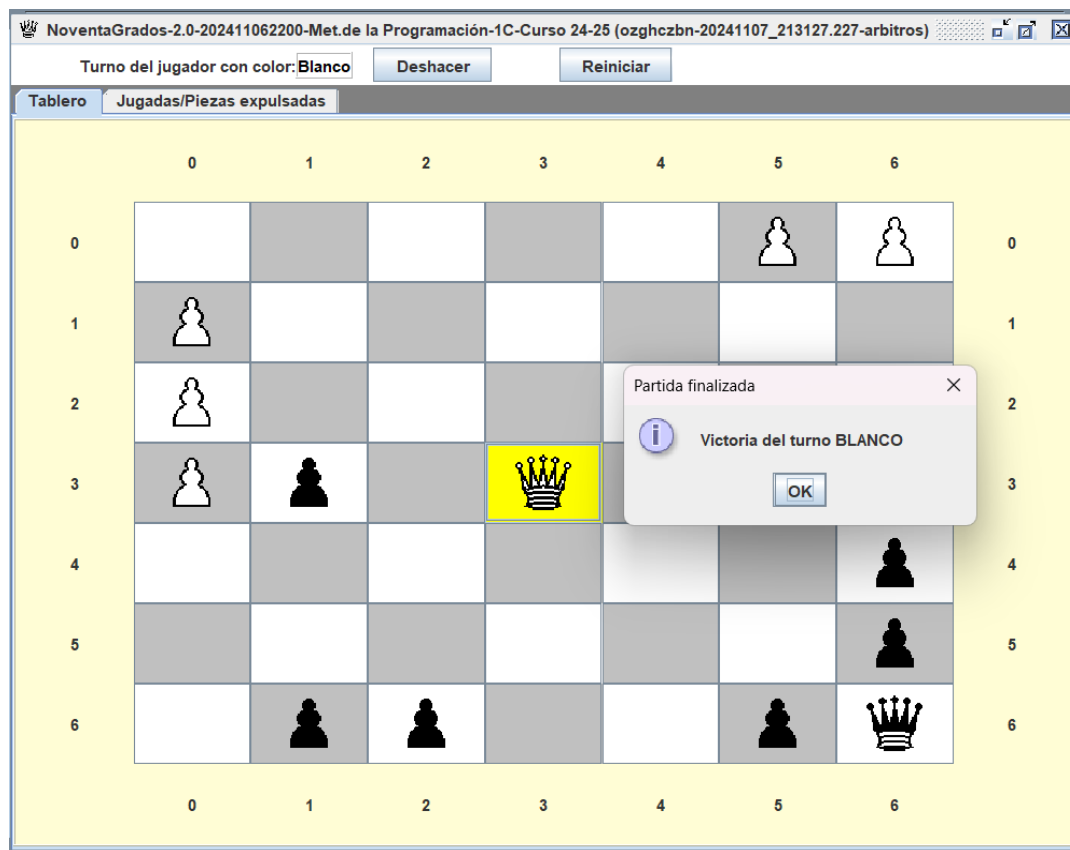


Ilustración 8: Partida finalizada (con victoria de blancas)

Las clases correspondientes a estos paquetes **se proporcionan en formato binario ya compiladas para la versión 22 de Java, en un fichero con nombre noventagrados-gui-lib-2.0.0.jar** y se debe reutilizar junto con los paquetes construidos previamente, configurando correctamente el classpath y **sin descomprimir en ningún caso el fichero .jar**. Se sugiere visualizar contenido del fichero .jar proporcionado, con un descompresor (e.g. 7Zip, WinZip, IZArc, tar, gzip, etc) aunque sin modificarlo, para aclarar conceptos de paquetes y clases.

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 - EPO2-1C**.

4.2 Formato de entrega

- Se enviará un fichero .zip, .rar, .7z o .tar.gz a través de la plataforma **UBUVirtual** completando la tarea **[MP] Entrega de Práctica Obligatoria 2 - EPO2-1C**.




- 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.7z`
 - Nombre `PrimerApellido-Nombre PrimerApellido.tar.gz`
- Ej: si los alumnos son María García y Pedro Fernández, su fichero `.zip` se llamará, sin utilizar tildes, `Maria Garcia-Pedro Fernandez.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 entregará individualmente por parte de cada uno de los miembros de la pareja a través de UBUVirtual. Verificad 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 cero.**
- **Cualquier situación de plagio detectado en las prácticas, conlleva la aplicación del Reglamento de Evaluación y del Reglamento de Régimen Disciplinario del Estudiantado de la UBU. Esto incluye también la utilización de código generado por herramientas basadas en inteligencia artificial.**
- **El alumnado es responsable de la privacidad de sus entregas, y en caso de detectarse su reutilización en entregas de segunda convocatoria, se aplicarán igualmente los reglamentos previamente indicados, aunque la asignatura ya haya sido superada en primera convocatoria.**

Se debe crear la siguiente estructura de directorios y ficheros en el **disco y entregar un único fichero con dicha estructura comprimida. Es obligatorio entregar un único fichero:**

- `/leeme.txt`: fichero de texto, que contendrá los nombres y apellidos y las aclaraciones que el alumnado crea oportunas poner en conocimiento del profesorado.
- `src`: ficheros fuentes (`.java`) y ficheros de datos necesarios para poder compilar el producto completo.
- `bin`: ficheros binarios (`.class`) generados al compilar.
- `doc`: documentación HTML generada con `javadoc` de todos los ficheros fuentes desarrollados por el alumnado.
- `/compilar.bat` o `/compilar.sh`: fichero de comandos con la invocación al compilador `javac` para generar el contenido de la carpeta `bin` a partir de los ficheros fuente en la carpeta `src`.
- `/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`.
- `/ejecutar_textui.bat` o `/ejecutar_textui.sh`: fichero de comandos con la invocación a la máquina virtual java para ejecutar la clase raíz del sistema en modo texto, utilizando las clases en el directorio `bin` y las bibliotecas necesarias en el directorio `lib`.
- `/ejecutar_gui.bat` o `/ejecutar_gui.sh`: fichero de comandos con la invocación a la máquina virtual java para ejecutar la clase raíz del sistema en modo gráfico, utilizando las clases en el directorio `bin` y las bibliotecas necesarias en el directorio `lib`.



-  **images:** se incluirán 3 capturas de la ejecución de la práctica, en modo gráfico, en **formato .jpg. o .jpeg. Incluyendo solo la pantalla completa del juego**, de manera similar a las ilustraciones incluidas en este enunciado (e.g. Ilustración 1, Ilustración 7 o Ilustración 8). Se pide:
 - 1 captura con la pantalla al inicio de la partida.
 - 1 captura con victoria de piezas blancas (discrecional).
 - 1 captura con victoria de piezas negras (discrecional).

El tamaño de cada imagen debería ser **inferior 150 KB**, si se supera este tamaño se recomienda usar una aplicación de compresión de imágenes (e.g. <https://www.iloveimg.com/es/comprimir-imagen> o <https://compressjpeg.com/es/>). Se recomienda hacer las capturas a una resolución máxima de pantalla de 1920 x 1080 píxeles y solo de la zona de interés. **El tamaño máximo de envío en UBUVirtual está limitado, impidiendo envíos de mayor tamaño.**

No se entregarán los directorios /lib ni /test ya proporcionados por los docentes. Se asume que los *scripts* entregados deben funcionar correctamente si se copian adicionalmente dichos directorios a las entregas realizadas.

Los *scripts* se pueden entregar bien para Windows (.bat) o bien para GNU/Linux o Mac OS X (.sh), pero se debe elegir **solo una plataforma**. Hay que tener en cuenta que en algunos *scripts* es necesario crear la carpeta de destino al principio.

Se recuerda que para los *scripts* de ejecución, se puede leer el primer argumento pasado a un *script* con %1 o \$1 respectivamente en Windows y GNU/Linux, para permitir pasar el valor *jugadas* o *arbitros*, sin necesidad de codificar dicho texto directamente dentro del *script*. Dicho argumento se le tendrá que pasar a la invocación de la máquina virtual.

Se deben revisar los ejemplos de *scripts* proporcionados en sesiones previas de prácticas, puesto que contienen parcialmente la solución. **Se deben probar previamente siempre antes de la entrega el correcto despliegue del producto 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 al profesorado 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 del alumnado** 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 (ver Anexo 2 y Anexo 3).
- **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 7,5 sobre un total de 15 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 recuerda la **prohibición del uso de más de 2 return** en métodos y del uso de **break y continue en bucles**. Esta regla no se aplica al código generado automáticamente por Eclipse (i.e., métodos `equals`, `hashCode` o `toString`).
- 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** tanto en el modo texto o gráfico, la calificación de la práctica estará **penalizada en un -40%**.
- Porcentajes/pesos⁴ aproximados en la valoración de la práctica:

Apartado	Cuestiones a valorar	Porcentaje (Peso)
Cuestiones generales de funcionamiento	Integrada la solución con la interfaz gráfica (5%). Documentada sin errores (3%). Scripts correctos (1%). Utiliza <code>package-info.java</code> . Extras de documentación (1%)	10,00%
NoventaGrados (textui)	Corrección del algoritmo propuesto. Correcto funcionamiento en modo texto. Correcto tratamiento de excepciones.	20,00%
Excepción comprobable	Correcta implementación con cuatro constructores.	2,50%
Mecanismo de deshacer	Interface correcta. Clase abstracta con atributos y métodos correctos. Correctas implementaciones de las clases concretas, con funcionamiento adecuado. Sin sobrecarga de atributos y correctas redefiniciones. Correcto tratamiento de excepciones. Correcto uso del <code>@Override</code>	30,00%
Arbitro	Uso de modificadores de acceso. Atributos correctos. Constructor e inicialización correcta. Colocación de piezas correctas. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de cambio de estado correctos. Métodos de consulta y cambio de estado correctos. Correcta gestión de jugadas y turno. Correcta implementación del empuje de piezas. Reglas correctamente implementadas. Resolución de finalización de partida. Correcta inclusión de genericidad.	15,00%
Caja	Correcta inicialización. Correcta clonación. Métodos de consulta correctos. Generados correctamente <code>equals</code> , <code>hashCode</code> y <code>toString</code> . Correcta inclusión de genericidad.	3,50%
TableroConsultor	Correcta implementación de métodos de consulta. Correcta inclusión de genericidad.	7,50%
Tablero	Uso de modificadores de acceso. Atributos correctos. Constructor correcto. Clonación correcta. Código no repetido en las implementaciones. Código no excesivamente largo en los métodos. Métodos de consulta correctos. Métodos de cambio de estado correctos. Uso de constantes simbólicas. Generados correctamente <code>equals</code> , <code>hashCode</code> y <code>toString</code> . Correcta inclusión de genericidad.	4,50%
Celda	Uso de modificadores de acceso. Atributos correctos. Constructores correctos. Métodos de consulta y cambio de estado correctos. Clonación correcta. Generados correctamente <code>equals</code> , <code>hashCode</code> y <code>toString</code> .	2,00%
Jugada	Tipo registro para la creación de jugadas con métodos correctos. Correcto método de consulta.	1,00%
Pieza	Uso de modificadores de acceso. Atributos correctos. Constructor correcto.	1,00%

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



	Clonación correcta. Métodos de consulta correctos. Generados correctamente equals, hashCode y toString.	
Color	Correcta implementación de enumeración, constructor, atributos y sus métodos asociados.	1,00%
Coordenada	Correcta implementación del tipo registro.	0,50%
Sentido	Correcta implementación de enumeración, constructor, atributos y sus métodos asociados.	1,00%
TipoPieza	Correcta implementación de enumeración, constructor, atributos y sus métodos asociados.	0,50%

Anexo 1. Generación de métodos equals, hashCode y toString en Eclipse

Los métodos `equals`, `hashCode` y `toString` son un estándar *de facto* en Java puesto que se heredan (y normalmente se redefinen) de la clase universal `Object`.

Para facilitar su implementación en la práctica, se utilizarán las **opciones** que ofrece **Eclipse** para su **generación automática**.

Este proceso debe realizarse **una vez completados los atributos de la clase**. Si estos cambiasen, es necesario **volver a repetir el proceso de generación** de dichos métodos (por lo tanto eliminar su código y volver a generarlos, salvo con los tipos registro o *record*, que precisamente resuelven toda esta problemática).

Para generarlos, se seleccionará el fichero fuente de la clase correspondiente y con el botón derecho, activando el menú contextual, se selecciona *Source* y *Generate hashCode() and equals()...* (ver Ilustración 9).

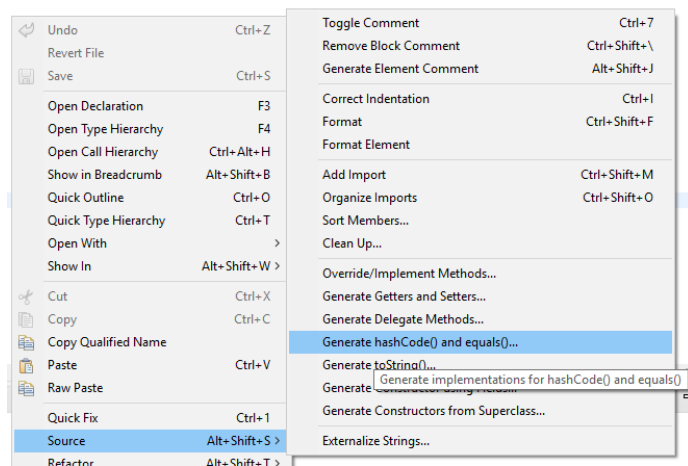


Ilustración 9: Generación de métodos hashCode y equals en Eclipse

A continuación se abre un diálogo mostrando los atributos a seleccionar para la generación de dichos métodos. Se marcan todos los atributos. Preferiblemente se indicará como "Insertion point:" el valor "Last member" para que el código **se añada al final**, dejando solo marcada la casilla para "Use Objects.hashCode and Object.equals methods (1.7 or higher)".

En la Ilustración 10, se muestra el ejemplo para la clase resuelta en la Sesión 4 del ejercicio del tres en raya `juego.modelo.Celda` (diferente de la clase `noventagraditos.modelo.Celda` solicitada en esta práctica) que contenía dos atributos `coordenada` y `pieza`.



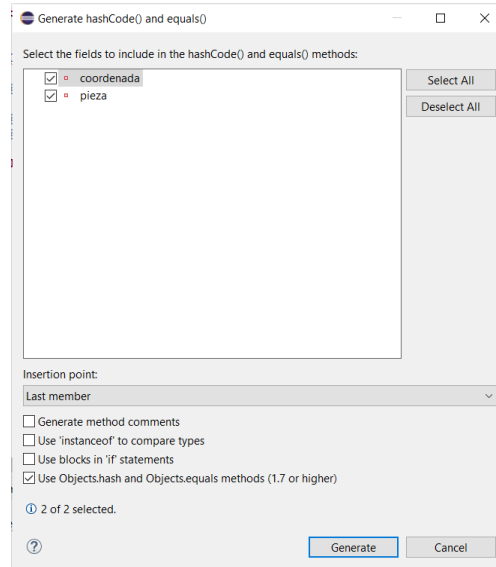


Ilustración 10: Opciones en la generación de los métodos

Para la generación del método `toString`, el proceso es similar, eligiendo en el menú contextual la opción *Generate toString()...* (ver Ilustración 11). Solo se seleccionan los atributos y se dejan el resto de opciones marcadas tal y como se muestran en dicha ilustración.

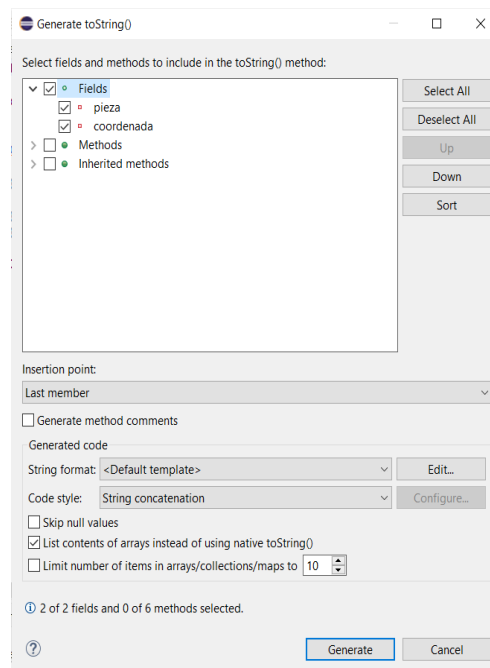


Ilustración 11: Generación del método toString

Finalizado este proceso, se tendrán ya implementados los **tres métodos**, según se requería en el enunciado y/o en el diagrama de clases correspondiente. Se recuerda que en estos métodos no se aplica la regla de limitación del número de **return** utilizados.



Anexo 2. Ejecución de pruebas automáticas con JUnit

En esta práctica se sigue utilizando **JUnit** como sistema de *testing*, aunque se usará alguna biblioteca adicional a las utilizadas en sesiones previas. Por lo tanto habrá que añadir dichas bibliotecas **al classpath (o build path en Eclipse)**, de manera similar a como se han añadido en sesiones previas de laboratorio.

En concreto en el directorio `lib`, se tendrán los siguiente ficheros que se proporcionan en UBUVirtual:

- `lib`
 - `noventagradados-gui-lib-2.0.0.jar*`
 - `hamcrest-all-1.3.jar`
 - `junit-platform-console-standalone-1.11.0.jar`

El fichero `noventagradados-gui-lib-2.0.0.jar`, es solo necesario para ejecutar la interfaz gráfica. No se utiliza en los *tests*.

Para la compilación y ejecución de los *tests* es necesario añadir tanto `hamcrest-all-1.3.jar`, como `junit-platform-console-standalone-1.11.0.jar` al `classpath` del sistema o "build path" de Eclipse.

Se recuerda que estos ficheros en la carpeta `lib`, **NO se deben adjuntar en la entrega final de la práctica.**

Anexo 3. Orden de implementación y ejecución ordenada de *suites*

Para orientar en el **orden de resolución** (implementación) **y testing**, se proporcionan varias *suites*. Se ha optado por utilizar la nomenclatura `SuiteLevelXTests` (donde `x` es un número) para indicar el orden en el que se deberían ejecutar las *suites* con los tests, según se vayan resolviendo las clases.

El orden sugerido de implementación para la resolución de la práctica con su consiguiente ejecución de los *tests* sería:

- `SuiteLevel1Tests`: prueba `Color`, `Coordenada`, `Sentido` y `TipoPieza`.
- `SuiteLevel2Tests`: prueba `Celda`, `Jugada` y `Pieza`
- `SuiteLevel3Tests`: prueba `Tablero`.
- `SuiteLevel4Tests`: prueba `Caja` y `TableroConsultor`.
- `SuiteLevel5Tests`: prueba `Arbitro` (*tests* de nivel básico).
- `SuiteLevel6Tests`: prueba `Arbitro` (*tests* de nivel medio).
- `SuiteLevel7Tests`: prueba `Arbitro` (*tests* de nivel avanzado).
- `SuiteLevel8Tests`: prueba la jerarquía de herencia en subpaquete `undo`.
- `SuiteLevel9Tests`: prueba que la excepción es comprobable.

Cada nivel incluye a los *tests* de los niveles previos, por ejemplo, cuando se ejecuta `SuiteLevel3Tests` se incluyen también los *tests* del nivel 2 y nivel 1.

En paralelo al uso de las *suites*, se pueden seguir ejecutando/depurando los *tests* de cada clase individualmente, seleccionado la clase con sufijo `Test` (e.g. si se selecciona `CeldaTest.java`, en el menú contextual se puede seleccionar *Run as... / JUnit Test* para ejecutar/depurar solo los *tests* relativos a la

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



clase `Celda`). Pero hay que ser consciente de que existen dependencias entre las clases, y no se deberían pasar a *tests* de nivel mayor, hasta no haber resuelto las clases más simples.

En el caso de `Arbitro`, se plantean *tests* en tres paquetes según el nivel de dificultad de la implementación a realizar. Se recuerda que también se puede ejecutar/depurar un test individualmente seleccionando en la vista JUnit.

Finalmente se pueden ejecutar **TODOS** los *tests*, utilizando `SuiteAllTests` (equivalente a ejecutar `SuiteLevel9Tests`).

Anexo 4. 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 `foreach`. 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 para 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.

Recursos

Bibliografía complementaria:

[Oracle, 2022] Lesson: Language Basics. The Java Tutorials (2022). Disponible en <http://docs.oracle.com/javase/tutorial/java/>.

[JDK22, 2024] JDK 20 Documentation. Disponible en <https://docs.oracle.com/en/java/javase/22/>

Enlaces a reglas del juego y vídeos demostrativos

90 GRAD -Tutorial en español y partida. (s. f.). BGG. <https://boardgamegeek.com/video/471392/90-grad/90-grad-tutorial-en-espanol-y-partida>

90 grad. (s. f.). BoardGameGeek. <https://boardgamegeek.com/boardgame/990/90-grad>. Vídeo de demostración del juego: <https://boardgamegeek.com/video/90746/90-grad/game-overview-recorded-at-spielwarenmesse-2016>



90 Grad - A fun & exciting journey into a beautiful & lush Marble Forrest | 90 Grad. (s. f.). BoardGameGeek.
<https://boardgamegeek.com/thread/1597644/90-grad-a-fun-and-exciting-journey-into-a-beautifu>

GAMES AND PUZZLES CALI. (2022, 21 marzo). *90 grados - 1001 juegos de mesa antiguos y nuevos en madera - Juegos del mundo* [Video]. YouTube. <https://www.youtube.com/watch?v=cYVYpOyNjd0>

iMisut, & iMisut. (2023, 14 diciembre). *Reseña: 90 Grad | Misut Meeple*. Misut Meeple | las Andanzas de un Amante de los Juegos de Mesa Modernos. <https://misutmeeple.com/2023/12/resena-90-grads/>

Mi juego mis reglas. (2023, 15 diciembre). *90 GRAD - Tutorial en español y partida // Gerhards Spiel und design* [Video]. YouTube. <https://www.youtube.com/watch?v=9a0xw1YtsAA>



Licencia

Autor: Raúl Marticorena Sánchez & Ismael Ramos Pérez & José Miguel Ramírez Sanz

Área de Lenguajes y Sistemas Informáticos

Departamento de Ingeniería Informática

Escuela Politécnica Superior

UNIVERSIDAD DE BURGOS

2024



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/>

