

PRÁCTICA 2

Codificación

UOC

Información relevante:

- Fecha límite de entrega: 17 de enero.
- Peso en la nota final de Prácticas: 70%.

Contenido

Información docente	3
Prerrequisitos	3
Objetivos	3
Resultados de aprendizaje	3
Introducción	4
Metodología en cascada o waterfall	4
Patrón Modelo-Vista-Controlador (MVC)	6
Enunciado	7
Entorno	7
Estructura de la práctica	7
Aspectos a tener en cuenta	8
Antes de empezar	9
Modelo	10
Controlador	15
Vistas	15
Corolario	18
Evaluación	19
Formato y fecha de entrega	20
Anexo: Soluciones a problemas	21

Información docente

Esta actividad pretende que pongas en práctica todos los conceptos relacionados con el paradigma de la programación orientada a objetos que has aprendido en la asignatura. En esta práctica la aplicación de dichos conceptos se llevará a cabo con la codificación del diagrama de clases UML de la Práctica 1.

Prerrequisitos

Para hacer esta Práctica necesitas:

- Tener asimilados los conceptos de los apuntes teóricos (i.e. los 4 módulos que se han tratado durante las PEC).
- Haber adquirido las competencias prácticas de las PEC. Para ello te recomendamos que mires las soluciones que se publicaron en el aula y las compares con las tuyas.
- Entender los elementos y conceptos básicos de un diagrama de clases UML.
- Tener asimilados los conocimientos básicos del lenguaje de programación Java trabajados durante el semestre. Para ello, te sugerimos repasar aquellos aspectos que consideres oportunos en la Guía de Java.

Objetivos

Con esta Práctica el equipo docente de la asignatura busca que:

- Sepas analizar un problema dado y codificar una solución a partir de un diagrama de clases UML y unas especificaciones, siguiendo el paradigma de la programación orientada a objetos.
- Te enfrentes a un programa de tamaño medio basado en un patrón de arquitectura como es MVC (Modelo-Vista-Controlador).
- Relaciones los conceptos de otras asignaturas previas con los de ésta.

Resultados de aprendizaje

Con esta Práctica debes demostrar que eres capaz de:

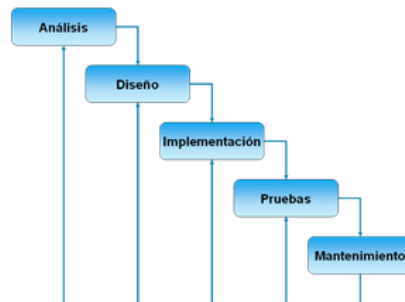
- Codificar un programa basado en un patrón de arquitectura como es MVC.
- Realizar algunas tareas relacionadas con la creación de interfaces gráficas para un programa Java.
- Emplear, de manera básica, la programación orientada a eventos.
- Usar ficheros de test en JUnit para determinar que un programa es correcto.
- Usar con cierta soltura un entorno de desarrollo integrado (IDE) como IntelliJ.

Introducción

El Equipo Docente considera oportuno que, llegados a este punto, relacionemos esta asignatura con conceptos propios de la ingeniería del software. Por ello, a continuación vamos a explicarte la metodología de desarrollo denominada “en cascada” (en inglés, *waterfall*) y el patrón de arquitectura software “Modelo-Vista-Controlador (MVC)”. Creemos que esta información, además de contextualizar esta Práctica, puede ser interesante para alguien que está estudiando una titulación afín a las TIC.

Metodología en cascada o *waterfall*

La metodología clásica para diseñar y desarrollar software se conoce con el nombre de metodología en cascada (o en inglés, *waterfall*). Aunque ha sido reemplazada por nuevas variantes, es interesante conocer la metodología original. En su versión clásica esta metodología está formada por 5 etapas secuenciales (ver siguiente figura).



La etapa de **análisis** de requerimientos consiste en reunir las necesidades del producto. El resultado de esta etapa suele ser un documento escrito por el equipo desarrollador (encabezado por la figura del analista) que describe las necesidades que dicho equipo ha entendido que necesita el cliente. No siempre el cliente se sabe expresar o es fácil entender lo que quiere. Normalmente este documento lo firma el cliente y es “contractual”.

Por su parte, la etapa de **diseño** describe cómo es la estructura interna del producto (p.ej. qué clases usar, cómo se relacionan, etc.), patrones a utilizar, la tecnología a emplear, etc. El resultado de esta etapa suele ser un conjunto de diagramas (p.ej. diagramas de clases UML, casos de uso, diagramas de secuencia, etc.) acompañado de información textual. Si nos decantamos en esta etapa por hacer un programa basado en programación orientada a objetos, será también en esta fase cuando usemos el paradigma *bottom-up* para la identificación de los objetos, de las clases y de la relación entre ellas.

La etapa de **implementación** significa programación. El resultado es la integración de todo el código generado por los programadores junto con la documentación asociada.

Una vez terminado el producto, se pasa a la etapa de **pruebas**. En ella se generan diferentes tipos de pruebas (p.ej. de test de integración, de rendimiento, etc.) para ver que el producto final hace lo que se espera que haga. Evidentemente, durante la etapa de

implementación también se hacen pruebas a nivel local –test unitarios (p.ej. a nivel de un método, una clase, etc.)– para ver que esa parte, de manera independiente, funciona correctamente.

La última etapa, **mantenimiento**, se inicia cuando el producto se da por terminado. Aunque un producto esté finalizado, y por muchas pruebas que se hayan hecho, siempre aparecen errores (*bugs*) que se deben ir solucionando a posteriori.

Si nos fijamos en la figura, siempre se puede volver atrás desde una etapa. Por ejemplo, si nos falta información en la etapa de diseño, siempre podemos volver por un instante a la etapa de análisis para recoger más información. Lo ideal es no tener que volver atrás.

En esta asignatura hemos pasado, de alguna manera, por las cuatro primeras fases. Así pues, la etapa de análisis la hemos hecho desde el Equipo Docente. Nosotros nos “hemos reunido” con el cliente y hemos analizado/documentado todas sus necesidades (Práctica 1). A partir de estas necesidades, hemos ido tomando decisiones de diseño. Por ejemplo, decidimos que el software se basaría en el paradigma de la programación orientada a objetos y que usaríamos el lenguaje de programación Java. Una vez decididos estos aspectos clave, hemos ido definiendo, a partir de la identificación de objetos, las diferentes clases (con sus atributos y métodos, i.e. datos y comportamientos) y las relaciones entre ellas a partir de las necesidades del cliente confirmadas en la etapa de análisis y de entidades reales que aparecen en el problema (i.e. objetos). Para ello hemos usado un paradigma *bottom-up*. Como puedes imaginar, la etapa de diseño es una de las más importantes y determinantes de la calidad del software, ya que en ella se realiza una descripción detallada del sistema a implementar. Es importante que esta descripción permita mostrar la estructura del programa de forma que su comprensión resulte sencilla. Por ese motivo es frecuente que la documentación de la fase de diseño vaya acompañada de diagramas UML, entre ellos los diagramas de clases (Práctica 1). Estos diagramas además de ser útiles para la implementación, también lo son en la fase de mantenimiento.

La etapa de implementación (o también conocida como desarrollo o codificación) la vamos a abordar en esta segunda práctica.

Finalmente, la etapa de **test/pruebas** la hemos tratado durante el semestre con los ficheros de test JUnit que se proporcionaban con los enunciados de las PEC y con alguna prueba extra que hayas hecho por tu cuenta. Estos ficheros nos permitían saber si las clases codificadas se comportaban como esperábamos. En esta segunda práctica, también ahondaremos en esta fase de testeo.

Evidentemente, la metodología en cascada no es la única que existe, hay muchas más: por ejemplo, prototipado y las actualmente conocidas como metodologías ágiles: eXtreme Programming, etc. En este punto podemos decir que en las PEC hemos seguido, en parte, una metodología TDD (*Test-Driven Development*), en cuya fase de diseño se codifican los test (te los hemos proporcionado con los enunciados) y son estos los que dirigen la fase de implementación. Si quieres saber más sobre metodologías para desarrollar software (donde

se incluyen los patrones) y UML, te animamos a cursar asignaturas de Ingeniería del Software. A estas metodologías de desarrollo de software se les debe añadir las estrategias o metodologías de gestión de proyectos, como SCRUM, CANVAS, así como técnicas específicas para la gestión de los proyectos, p.ej. diagramas de Gantt y PERT, entre otros.

Patrón Modelo-Vista-Controlador (MVC)

En esta Práctica usaremos el patrón de arquitectura de software llamado MVC (Model-View-Controller, i.e. modelo, vista y controlador). El patrón MVC es muy utilizado en la actualidad, especialmente en el mundo web. De hecho, con el tiempo han surgido variantes como MVP (P de Presenter) o MVVM (Modelo-Vista-VistaModelo). En líneas generales, MVC intenta separar tres elementos clave de un programa:

- **Modelo:** se encarga de almacenar y manipular los datos del programa. En la mayoría de ocasiones esta parte recae sobre una base de datos y las clases que acceden a ella. Así pues, el modelo se encarga de realizar las operaciones CRUD (i.e. Create, Read, Update y Delete: crear, consultar, actualizar y eliminar) sobre la información del programa, así como de controlar los privilegios de acceso a dichos datos. Una alternativa a la base de datos es el uso de ficheros de texto y/o binarios.
- **Vista:** es el conjunto de “pantallas” que configura la interfaz con la que interactúa el usuario. Cada “pantalla” o vista puede ser desde una interfaz por línea de comandos hasta una interfaz gráfica, diferenciando entre móvil, tableta, ordenador, etc. Cada vista suele tener una parte visual y otra interactiva. Esta última se encarga de recibir los inputs/eventos del usuario (p.ej. clic en un botón) y de comunicarse con el/los controlador/es del programa para pedir información o para informar de algún cambio realizado por el usuario. Además, según la información recibida por el/los controlador/es, modifica la parte visual en consonancia.
- **Controlador:** es la parte que controla la lógica del negocio. Hace de intermediario entre la vista y el modelo. Por ejemplo, mediante una petición del usuario (p.ej. hacer clic en un botón), la vista –a través de su parte interactiva– le pide al controlador que le dé el listado de personas que hay almacenadas en la base de datos; el controlador le solicita esta información al modelo, el cual se la proporciona; el controlador envía la información a la vista que se encarga de procesar (i.e. parte interactiva) y mostrar (i.e. parte visual) la información recibida por el controlador.

Gracias al patrón MVC se desacoplan las tres partes. Esto permite que teniendo el mismo modelo y el mismo controlador, la vista se pueda modificar sin verse alteradas las otras dos partes. Lo mismo, si cambiamos el modelo (p.ej. cambiar de gestor de base de datos de MySQL a Oracle), el controlador y las vistas no deberían verse afectadas. Lo mismo si modificáramos el controlador. Así pues, con el uso del patrón MVC se minimiza el impacto de futuros cambios y se mejora el mantenimiento del programa. Si quieres saber más, te recomendamos ver el siguiente vídeo: <https://youtu.be/UU8AKk8Slqg>.

Enunciado

En esta Práctica vas a codificar el programa explicado en el enunciado de la Práctica 1.

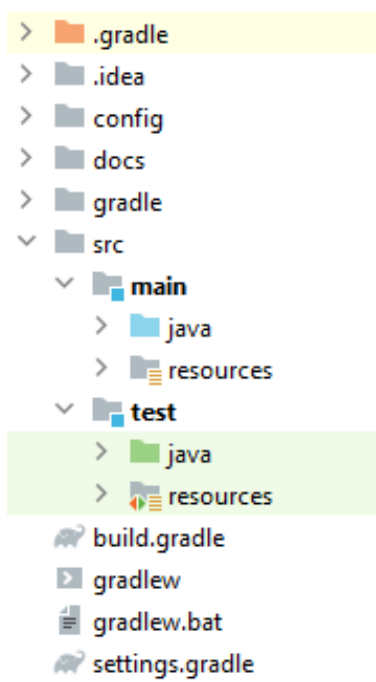
Entorno

Para esta práctica utiliza el siguiente entorno:

- JDK ≥ 15 .
- IntelliJ Community.
- Gradle, quien descargará las dependencias necesarias para el proyecto.

Estructura de la práctica

Si abres el .zip que se te proporciona con este enunciado, encontrarás el proyecto UOCTrip. Si lo abres en IntelliJ, verás la estructura que se muestra en la siguiente imagen. De dicha estructura cabe destacar:



docs: contiene la documentación Javadoc del proyecto finalizado. Abre el fichero `index.html` en un navegador web para ver la documentación del programa. Esta información complementa las explicaciones dadas en este enunciado.

src: es el proyecto en sí, el cual sigue la estructura de directorios propia de Gradle (y Maven).

En `src/main/java` verás tres paquetes llamados `model` (dividido en 3 subpaquetes), `views` (dividido en `cmd` y `gui`) y `controller`. Lo hemos organizado así porque, como hemos comentado, usaremos el patrón MVC.

En `src/main/resources` encontrarás las imágenes y pantallas que se utilizan en la vista gráfica del juego así como los ficheros de configuración de los niveles.

Por su parte, `src/test/main` contiene los ficheros de test JUnit. Asimismo, en `src/test/resources` encontrarás ficheros de configuración de niveles que son utilizados para testear el programa.

config: en esta Práctica usarás Checkstyle, lo que te permitirá escribir un código que cumpla con los estándares y buenas prácticas de estilo de Java. En este directorio está el fichero de configuración. Hablamos de Checkstyle en el apartado de Evaluación.

build.gradle: este fichero contiene toda la configuración necesaria de Gradle. En él hemos definido tareas específicas para esta Práctica con la finalidad de ayudarte durante la realización de la misma.

Aspectos a tener en cuenta

En este apartado queremos resaltar algunas cuestiones que consideramos importantes que tengas presentes durante la realización de la Práctica.

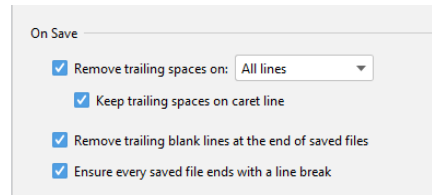



1. Para hacer esta práctica, además de la información del diagrama de clases que te proporcionamos para el modelo, **deberás tener en cuenta las especificaciones que se indiquen en este enunciado y en el Javadoc que se proporciona (directorio docs).**
2. Aunque es una buena práctica, **no es necesario que utilices comentarios Javadoc en el código ni que tampoco generes la documentación del programa.**
3. Durante la realización de la Práctica **puedes añadir todos los atributos y métodos que quieras.** La única condición es que deben ser **declarados con el modificador de acceso `private`.**
4. Puedes usar cualquier clase, interfaz y/p enumeración que te proporcione la API de Java. Sin embargo, **no puedes añadir dependencias** (i.e. librerías de terceros) que no se indiquen en este enunciado.
5. Antes de seguir leyendo, te recomendamos **leer los criterios de evaluación** de esta Práctica.
6. Cuando tengas problemas, relee el enunciado, busca en los apuntes de la asignatura y en Internet. Si aun así no logras resolverlas, **usa el foro del aula antes que el correo electrónico.** Piensa que tu duda la puede tener otro compañero. Eso sí, **en el foro no se puede compartir código.**
7. **La realización de la Práctica es individual.** Si se detecta el más mínimo indicio de plagio/copia, el Equipo Docente se reserva el derecho de poder contactar con el estudiante para aclarar la situación. Si finalmente se ratifica el plagio, la asignatura quedará suspendida con un 0 y se iniciarán los trámites correspondientes para abrir un expediente sancionador, tanto para el estudiante que ha copiado como para el que ha facilitado la información .
8. **Los test proporcionados no deben ser modificados.** Asimismo, cualquier práctica en la que se detecten trampas, p.ej. *hardcodear* código para que supere los test, será suspendida con un 0.
9. **La fecha límite indicada en este enunciado es inaplazable.** Puedes hacer diversas entregas durante el período de realización de la Práctica. El Equipo Docente corregirá la última entrega. Asegúrate de que entregas los ficheros correctos. **Una vez finalizada la fecha límite, no se aceptarán nuevas entregas.**

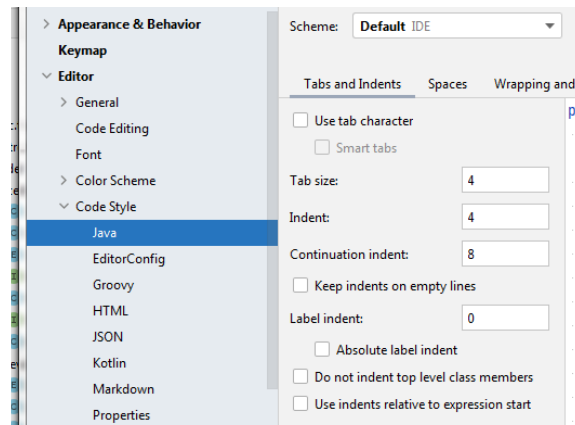
Antes de empezar

Antes de codificar, queremos que te asegures de que tienes IntelliJ configurado como se indica en este apartado. Ves a **File** → **Settings...**

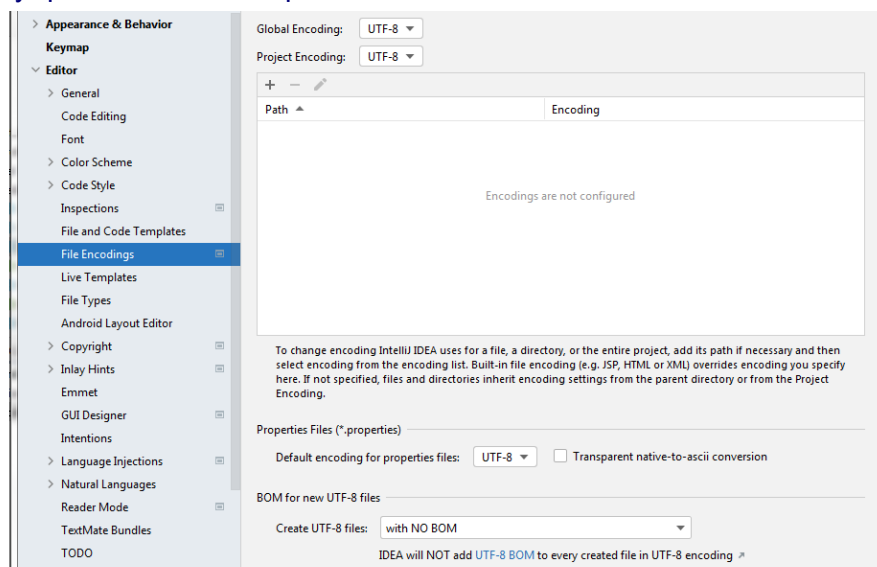
En la ventana escoge **Editor** → **General**. En la parte de la derecha, baja y marca todos los ítems del apartado **On Save**.



Ahora escoge, en el menú izquierdo, **Editor** → **Code Style** → **Java**. En la parte de la derecha asegúrate de que en **Tabs and Indents** (si no aparece, haz click en el icono  de la derecha) tienes desmarcada la opción **Use tab character**.



En el menú izquierdo elige **Editor** → **File Encodings**. Asegúrate que todos los valores sean **UTF-8** y que el valor de la opción **Create UTF-8 files** es **"with NO BOM"**.



Modelo

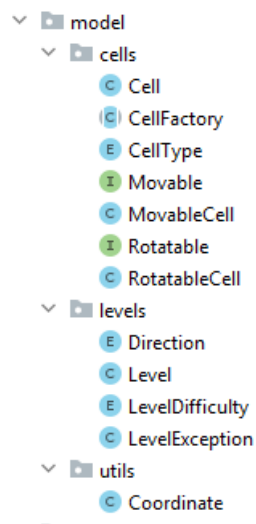
En la Práctica 1 se te pidió que hicieras el diagrama de clases UML del modelo del programa que vamos a codificar en esta segunda práctica:



Con este enunciado te proporcionamos el diagrama de clases UML del modelo que vamos a utilizar. Es la solución de la Práctica 1 con algún añadido/modificación.

A continuación vamos a guiarte en la codificación del modelo, dándote información adicional que creemos necesaria e intentando que codifiques el modelo desde lo más sencillo a lo más difícil (aunque no siempre será posible por la dependencia entre elementos).

Antes de nada ten en cuenta que el paquete `model` se estructurará de la siguiente manera:



Siendo `cells`, `levels` y `utils`, tres subpaquetes de `model`.

Clase `CellFactory`

Esta clase es nueva y no aparece en el diagrama de clases, pero **te la damos ya codificada y no tienes que hacer nada con ella**. La hemos definido para poder seguir un patrón denominado Factoría, de ahí su nombre. Este patrón es usado principalmente cuando tenemos una clase/interfaz con muchas subclases o implementaciones y según un *input* en tiempo real necesitamos devolver una de estas subclases/implementaciones concretas. No es parte de esta asignatura conocer este patrón. De hecho, nosotros hemos codificado, por la sencillez del programa, un *simple factory*, el cual muchos programadores no consideran un patrón. Aquí tienes un vídeo que explica *simple factory* en Java: <https://www.youtube.com/watch?v=3iWDjpwJAag>.

Las variantes *factory* sí consideradas patrones son *method factory* y *abstract factory*. Aquí tienes, por si quieres ampliar conocimientos, un vídeo explicando *method factory* basado en videojuegos: <https://www.youtube.com/watch?v=ILvYAzXO7Ek>.

En este vídeo se explican las diferencias entre *method factory*, *abstract factory* y *simple factory*: https://www.youtube.com/watch?v=KS5_FVxmTX8.

Una explicación rápida y sencilla de las tres variantes de factoría la puedes leer en: <https://vivekcek.wordpress.com/2013/03/17/simple-factory-vs-factory-method-vs-abstract-factory-by-example/>.

Interfaz Movable

Codifica esta interfaz siguiendo las indicaciones del diagrama de clases.

Interfaz Rotatable

Codifica esta interfaz siguiendo las indicaciones del diagrama de clases.

Enumeración LevelDifficulty

Codifica esta enumeración siguiendo las indicaciones del diagrama de clases.

Clase LevelException

Codifica esta clase heredando de la clase de Java `Exception` y siguiendo las indicaciones del diagrama de clases.

Enumeración Direction

Codifica esta enumeración siguiendo las indicaciones del diagrama de clases. Asimismo ten en cuenta que el objetivo de esta enumeración es recopilar las direcciones en las que el “coche” podrá desplazarse, i.e. arriba, derecha, abajo e izquierda. Este orden de las direcciones es un estándar *de facto*. Por ejemplo, si para un cuadrado definimos en CSS (permite definir estilos en páginas web) lo siguiente: `border: 1px 2px 3px 4px solid red;` estaríamos diciendo que los bordes de dicho cuadrado son líneas rojas con un grosor de 1px para la parte de arriba, 2px el lado derecho, 3px el lado inferior y 4px el lado izquierdo.

Como verás en el diagrama de clases, cada dirección es “construido” a partir de tres parámetros: (1) `dRow` indica el desplazamiento (i.e. número de casillas) en vertical que se hace en dicha dirección; (2) `dColumn` indica el desplazamiento en horizontal que se hace en dicha dirección, y (3) `opposite` indica el índice del valor de la enumeración `Direction` que es el opuesto de la dirección actual. Así pues:

Valor	Valor para dRow	Valor para dColumn	Valor para opposite
UP	-1	0	2
RIGHT	0	+1	3
DOWN	+1	0	0
LEFT	0	-1	1



Pista: Recuerda que los valores de la enumeración tienen un índice interno que sigue el orden de declaración. En este caso, el valor `UP` ocupa el índice 0 y el valor `LEFT` ocupa el índice 3. Saber esto te ayudará a la hora de codificar los métodos `getValueByIndex` y `getOpposite`.

Clase Coordinate

Codifica esta clase según las indicaciones del diagrama de clases. También ten en cuenta:

- **equals:** la sobrescritura de este método debe devolver `true` cuando las dos coordenadas sean la misma, es decir, cuando sus filas coincidan y sus columnas también. En caso contrario (i.e. no coincidencia de las filas y/o columnas, una coordenada es `null` o se pasa como argumento un objeto que no sea del tipo `Coordinate`), debe devolver `false`.
- **hashCode:** cuando se sobrescribe el método `equals`, la documentación de Java nos dice que es una buena práctica sobrescribir también el método `hashCode` de la clase `Object` de manera que dos objetos que sean iguales (i.e. `equals` devuelve `true`) tengan el mismo valor `hash` (no es más que un `int`). Para hacer esto, debes usar el método estático `hash` de la clase `Objects` y pasarle como parámetros los valores de la fila y la columna, en este orden (para que no falle el test).
- **toString:** sobrescribe este método de manera que devuelva el siguiente texto "`(r,c)`", siendo `r` el valor de la fila y `c` el valor de la columna, p.ej. "`(5,4)`".

Enumeración CellType

Codifica esta enumeración siguiendo las indicaciones del diagrama de clases. Asimismo ten en cuenta que:

- **connections:** es un `array` de `boolean` que dice si en una dirección tiene entrada/salida (`true`) o no (`false`). Seguiremos el mismo estándar que con `Direction`, siendo la primera casilla del `array` el valor para arriba, siguiéndole, derecha, abajo e izquierda.
- **setConnections:** asigna el nuevo `array` al atributo `connections` de `CellType`, reemplazando lo que había. Lo hace asignando todo el `array` de una vez, es decir, no asigna valor a valor al `array` ya existente.

- **map2CellType**: dado un carácter que se usa en los ficheros de configuración de los niveles, devuelve el valor de la enumeración `CellType` que le corresponde. Si el carácter no corresponde a ningún valor de `CellType`, entonces devuelve `null`.
- **getAvailableConnections**: devuelve un `EnumSet<Direction>` que contiene todas las direcciones (i.e. valores de la enumeración `Direction`) en las que la pieza/celda tiene conexión (i.e. entrada/salida).
- **next**: este método es abstracto y será definido por cada valor de la enumeración `CellType` de manera que devuelva el siguiente elemento si éste fuera rotado 90° hacia la izquierda (i.e. en el sentido de las agujas del reloj):

Valores	Valor a devolver por next
START FINISH MOUNTAINS RIVER FREE	<code>null</code>
VERTICAL	HORIZONTAL
HORIZONTAL	VERTICAL
BOTTOM_RIGHT	BOTTOM_LEFT
BOTTOM_LEFT	TOP_LEFT
TOP_RIGHT	BOTTOM_RIGHT
TOP_LEFT	TOP_RIGHT
ROTATABLE_VERTICAL	ROTATABLE_HORIZONTAL
ROTATABLE_HORIZONTAL	ROTATABLE_VERTICAL



Pista: Lee en el apartado 3.13.7 de la Guía de Java cómo funcionan los métodos abstractos en las enumeraciones.

Clase `Cell`

Codifica esta clase siguiendo las indicaciones del diagrama de clases.

Clase `RotatableCell`

Codifica esta clase siguiendo las indicaciones del diagrama de clases. Ten presente que el método `rotate` rota el elemento 90° en sentido horario.

Clase MovableCell

Codifica esta clase siguiendo las indicaciones del diagrama de clases. Ten presente que el método `move` reemplaza la coordenada de la celda por la coordenada que se pasa como argumento.

Clase Level

Para esta clase te damos algunos métodos ya hechos que no debes modificar. Vamos a explicar el resto:

- **getSize:** *getter* del atributo `size`.
- **setSize:** *setter* del atributo `size` que debe lanzar una excepción `LevelException` con el mensaje de `ERROR_BOARD_SIZE` cuando el nuevo valor es menor a 3. En caso de lanzar una excepción, el nuevo valor no debe asignarse.
- **getDifficulty y setDifficulty:** *getter* y *setter* del atributo `difficulty`, respectivamente. No deben hacer ninguna comprobación.
- **getNumMoves y setNumMoves:** *getter* y *setter* del atributo `numMoves`, respectivamente. No deben hacer ninguna comprobación.
- **validatePosition:** devuelve `true` si la coordenada está dentro del rango `[0, size)` tanto para la fila como para la columna. En caso contrario, devuelve `false`. Si el objeto pasado como parámetro es `null`, también devuelve `false`.
- **getCell:** si la posición es válida, entonces devuelve el objeto `Cell` que hay en la coordenada indicada como parámetro. En caso contrario, lanza una `LevelException` con el mensaje de `ERROR_COORDINATE`.
- **setCell:** si la posición es válida, entonces asigna en dicha coordenada del tablero el objeto `Cell` pasado como parámetro. En caso contrario o si el objeto `Cell` es `null`, lanza una `LevelException` con el mensaje de `ERROR_COORDINATE`.
- **swapCells:** si las dos piezas que hay en las coordenadas indicadas como parámetro son móviles, entonces este método las intercambia en el tablero e incrementa un unidad el número de movimientos realizados. En caso contrario, lanza una `LevelException` con el mensaje de `ERROR_NO_MOVABLE_CELL`.
- **rotateCell:** si la pieza que hay en la coordenada indicada como parámetro es rotable, entonces la rota una vez e incrementa un unidad el número de movimientos realizados. En caso contrario, lanza una `LevelException` con el mensaje de `ERROR_NO_ROTATABLE_CELL`.
- **isSolved:** devuelve `true` si existe un camino correcto entre la casilla inicial y la final. Ten presente que la casilla inicial sólo tiene conexión por arriba y la casilla final sólo por abajo. Asimismo, la casilla inicial está en la última fila y la final en la primera.
- **toString:** devuelve un `String` cuyo contenido es el tablero con las piezas en formato unicode y con los valores de columna (1, 2, etc.) y fila (a, b, etc.). La representación sería la que se mostró en la Figura 2 del enunciado de la Práctica 1.

Controlador

El controlador es quien maneja la lógica del negocio. En este caso, la lógica del videojuego. Es decir, el controlador es el responsable de decidir qué hacer con la petición que ha realizado el usuario desde la vista. Lo habitual es hacer una petición al modelo.

En el paquete `controller` del proyecto verás una clase llamada `Game`. Ésta es la clase controladora del juego (un programa puede tener varias clases controladoras).

Clase `Game`

Te proporcionamos el esqueleto de esta clase y algún método ya codificado que no debes modificar. El resto de métodos tienen el comentario `//TODO` y los debes codificar siguiendo las especificaciones indicadas en el Javadoc. Para encontrar rápido dónde hay un comentario `//TODO` en tu código, puedes ir a `View → Tool Windows → TODO`. Mostrará una pestaña `TODO` en la parte inferior con todos los sitios del código donde hay `//TODO`.

Vistas

Las vistas son las “pantallas” con las que interactúa el usuario. En este caso, tenemos dos maneras de interactuar: (1) textual basada en línea de comandos, y (2) gráfica.

Vista textual (`view.cmd`)

La vista textual se encuentra en el paquete `view.cmd` y te la damos codificada. De esta manera, puedes verificar si el juego funciona o, dicho de otro modo, si has codificado correctamente tanto el modelo como el controlador. Para ejecutar esta vista, verás que la ventana de Gradle de IntelliJ muestra dentro de `execution` una tarea llamada `runCmdVersion`. Si ejecutas esta tarea, se ejecutará el juego en modo textual. La vista textual se ejecuta por consola. En cada jugada, el jugador debe escribir, con el formato `filaColumna` (p.ej. `a1`), la casilla que desea intercambiar o rotar, y a continuación debe presionar enter. Seguidamente, si la pieza es móvil, la vista pedirá al jugador, en el mismo formato de antes, la casilla con la que hay que hacer el intercambio y a continuación debe presionar enter. Con cada acción, se repinta la pantalla con los cambios producidos.

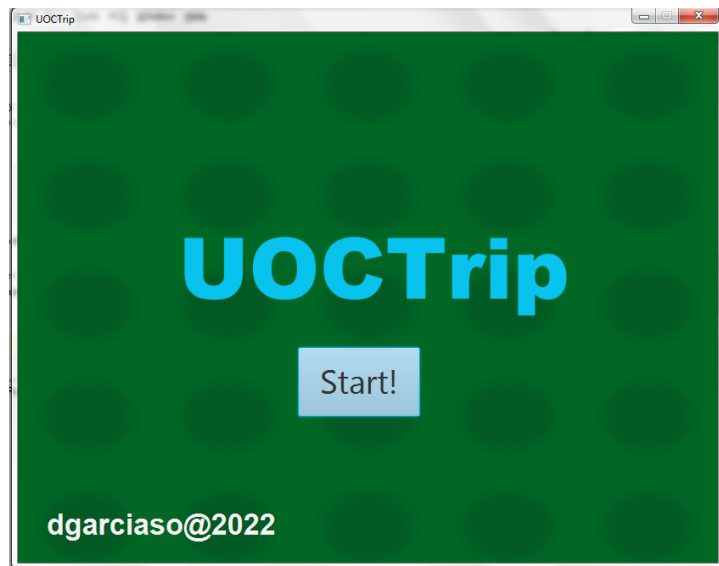
Vista gráfica (`view.gui`)



Para hacer esta parte del enunciado te recomendamos leer el apartado 5.2 de la Guía de Java obviando las referencias a Eclipse. Como verás se sugiere usar el programa Scene Builder, el cual permite crear y modificar interfaces gráficas de manera WYSIWYG. Encontrarás Scene Builder en el siguiente enlace: <https://gluonhq.com/products/scene-builder/#download>. Si quieres vincular Scene Builder con IntelliJ (no es obligatorio, pero sí práctico), ves a `File → Settings...` Luego a `Languages & Frameworks`. Dentro escoge la opción `JavaFX` y en el lado derecho indica dónde está el fichero ejecutable de Scene Builder. A partir de aquí, podrás hacer clic derecho en IntelliJ sobre un fichero `.fxml` y decirle que lo abra con Scene Builder. IntelliJ también muestra para los ficheros `.fxml` una pestaña "Scene Builder" que integra Scene Builder dentro del IDE.

Con el proyecto ya te damos las vistas/pantallas del programa hechas. No queremos que pierdas demasiado tiempo creando interfaces (no estás en una asignatura de diseño gráfico y usabilidad). Decimos “demasiado tiempo” porque se te pide que en el paquete `view.gui`:

- a) Añadas en la pantalla `Welcome.fxml` que está ubicado en `src/main/resources/fxml`, un texto con tu login UOC seguido de “@2022”. Este texto debe ser Arial, 34px., negrita y de color blanco (ver la siguiente imagen).



- b) Añadas un botón con el texto "Reload" en la pantalla `Play.fxml` que está ubicada en `src/main/resources/fxml` (ver la siguiente imagen).



Ahora vamos a abordar la parte interactiva.

- c) Al botón "Reload" que has añadido en el apartado B, asígnale la funcionalidad de que al dejar de ser clicado (*on release*) reinicie el nivel (i.e. como si empezara a jugar de nuevo ese nivel).



Pista: Para codificar esta funcionalidad que se te pide en el apartado C, deberás seguir el patrón MVC. Concretamente tendrás que crear un método en la parte interactiva para que se ejecute cada vez que se pulse en el botón "Reload" y que, a su vez, este método se comunique con el controlador de la aplicación para que ejecute el método `reload`, quien realmente hará toda la tarea.

El resto del programa en formato gráfico ya viene codificado. Para ejecutar esta vista, verás que la ventana de Gradle de IntelliJ muestra dentro de `execution` una tarea llamada `runGuiVersion`. Si ejecutas esta tarea, se ejecutará el juego en modo gráfico. La manera de jugar es clicando en una pieza y después, si es una pieza móvil, en la pieza con la que se quiere intercambiar. Si la pieza es rotable, sólo hay que hacer clic en la pieza en cuestión.

Corolario

Si estás leyendo esto, es que ya has terminado la Práctica 2. ¡¡Felicidades!! Llegados a este punto, seguramente te estés preguntando: *¿cómo hago para pasarle el programa a alguien que no tenga ni IntelliJ ni JDK instalados?* Buena pregunta. La respuesta es que debes crear un archivo ejecutable, concretamente, un JAR (Java ARchive). Un `.jar` es un tipo de fichero –en verdad, un `.zip` con la extensión cambiada– que permite, entre otras cosas, ejecutar aplicaciones escritas en Java. Gracias a los `.jar`, cualquier persona que tenga instalado JRE (*Java Runtime Environment*) lo podrá ejecutar como si de un fichero ejecutable se tratase. Normalmente, los ordenadores tienen JRE instalado.

Para crear un fichero `.jar` para una aplicación JavaFX hay que tener presente que la clase principal (i.e. aquella que tiene el `main`) no puede heredar de `Application`. Si lo hace, el `.jar` no se ejecutará correctamente. Es por ello que la solución más sencilla es crear una nueva clase que llame al `main` de la clase que hereda de `Application`. Si miras el fichero `build.gradle`, verás que la tarea `runGuiApp` usa como `main`, el que tiene `GuiApp`, mientras que `jar` invoca al `main` de la clase `Main`. Asimismo, debido a que JavaFX no pertenece al *core* de JDK desde la versión 11, debemos añadir los módulos que el programa necesita, de lo contrario, la ejecución del `.jar` fallará. Para indicarle los módulos debemos hacer el proyecto modular, que no es más que añadir el fichero `module-info.java` al proyecto. Si te fijas, te hemos facilitado dicho fichero en `src/main/java`.



Gracias a Gradle sólo tienes que hacer doble click en la tarea `jar` y se creará el fichero `.jar` dentro de una carpeta llamada `build`. Más concretamente, está dentro de `build/libs`. Simplemente copia el fichero `UOCTrip-1.0.jar` (contiene todo: ficheros e imágenes) y ejecútalo donde quieras (asegúrate que en el ordenador que utilices esté, como mínimo, la versión 15 de JRE). Puedes ejecutarlo haciendo doble click o usando el comando `java -jar UOCTrip-1.0.jar` en un terminal.

Quizás estés pensando: *¿qué sucede si en el ordenador en que se ejecuta el `.jar` no hay JRE ni JDK?* Pues, o bien lo instalas, o bien usas `jlink`. Si quieres probarlo, puedes añadir el *plugin* `jlink` en `build.gradle`. No obstante, te avisamos que con aplicaciones que leen ficheros (como `.txt` o imágenes) da bastantes problemas. Lo que hace `jlink` es empaquetar el `.jar` junto con una versión *ad hoc* de JRE. Para ello necesita que el proyecto Java esté modularizado, puesto que, según los módulos que se indiquen en el fichero `module-info.java`, el JRE *ad hoc* que cree será mayor o menor. Hoy en día se usan aplicaciones como Docker para distribuir programas.

¿Y si queremos un instalador? Pues a partir de JDK 16 está disponible `jpackage`. Lee más sobre `jar`, `jlink` y `jpackage` en: <https://dev.to/cherrychain/javafx-jlink-and-jpackage-h9>.

Evaluación

Esta Práctica se evalúa de la siguiente manera:

Elemento	Peso	Comentarios
39 test etiquetados como "sanity"	20%	Gradle → verification → testSanity Estos test deben ser pasados satisfactoriamente en su totalidad para lograr el 20%. Estos test aseguran que la parte crítica del esqueleto del programa es respetada. Por ello, si alguno de estos test falla, entonces la nota que obtendrás en la Práctica 2 será, como máximo, un 1.
36 test etiquetados como "minimum"	30%	Gradle → verification → testMinimum Estos test deben ser pasados satisfactoriamente en su totalidad para lograr el 30%. Si superas todos los test "sanity", pero alguno de los test "minimum" falla, entonces la nota que obtendrás en la Práctica 2 será, como máximo, un 3.
 Gradle → verification → testPassPractice ejecuta los test "sanity" y "minimum". Si los 75 test de testPassPractice son superados, tu Práctica tendrá, como mínimo, una calificación de 5. Si superas todos los test de testPassPractice, entonces los siguientes test e ítems serán evaluados de manera aislada según indica la columna "comentarios".		
19 test etiquetados como "advanced"	20%	Gradle → verification → testAdvanced (# test pasados / # test) * 20% No sirve poner <code>return true/false</code> en <code>isSolved</code> .
2 test etiquetados como "controller"	10%	Gradle → verification → testController (# test pasados / # test) * 10%
 Gradle → verification → testAll ejecuta todos los test anteriores. Si se superan los 96 test, la nota de la Práctica es, como mínimo, un 8.		
Calidad <code>isSolved</code>	5%	Calidad del algoritmo del método <code>isSolved</code> .
Vista	10%	Poner texto = 2%; Poner botón = 2%; Funcionalidad botón = 6%
Estilo de código	5%	Gradle → other → checkstyleMain Tras ejecutar Checkstyle, habrá un informe en <code>build/reports/checkstyle/main.html</code> . Ábrelo con un navegador web. Tendrás el 5% de inicio, pero por cada 5 errores de Checkstyle te será descontado un 1%.

Formato y fecha de entrega

Tienes que entregar un fichero *.zip, cuyo nombre tiene que seguir este patrón: loginUOC_PRAC2.zip. Por ejemplo: dgarciaso_PRAC2.zip. Este fichero comprimido tiene que incluir los siguientes elementos:

- El proyecto `UOCTrip` completado siguiendo las peticiones y especificaciones del enunciado.



Antes de entregar, ejecuta la tarea de Gradle `build → clean` que borrará el directorio `build`.

El último día para entregar esta Práctica es el **17 de enero de 2022** antes de las 23:59. Cualquier Práctica entregada más tarde será considerada como no presentada.

Anexo: Soluciones a problemas

1. Si en la consola tienes problemas de *encoding* (i.e. que algún carácter no se muestra correctamente), sigue las indicaciones que se dan en el siguiente enlace: <https://www.jetbrains.com/help/idea/encoding.html#console>.
2. Si no funciona JavaFX, asegúrate de que tienes asociado al proyecto, como mínimo, la versión 11 de JDK. Para más información de cómo configurar JavaFX con Gradle en IntelliJ, consulta el siguiente enlace: <https://openjfx.io/openjfx-docs/#IDE-IntelliJ>.