

PRÁCTICA 2

Codificación

UOC

Información relevante:

- Fecha límite de entrega: 5 de julio.
- Peso en la nota final de Prácticas: 70%.

Contenido

Información docente	3
Prerrequisitos	3
Objetivos	3
Resultados de aprendizaje	3
Introducción	5
Metodología en cascada o waterfall	5
Patrón Modelo-Vista-Controlador (MVC)	7
Enunciado	8
Entorno	8
Estructura de la práctica	8
Aspectos a tener en cuenta	10
Antes de empezar	11
Modelo (8 puntos)	12
Sugerencia de orden a seguir a la hora de codificar	13
Indicaciones	14
Controlador (1.5 puntos)	18
Vistas (0.5 puntos)	18
Corolario	20
Criterios de evaluación	21
Formato y fecha de entrega	23

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 programa planteado en 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.
- Utilizar un framework de desarrollo de videojuegos para Java..
- 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.

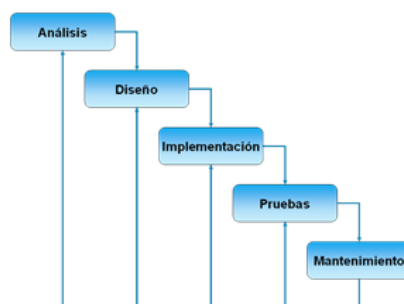


Introducción

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 hemos trabajado durante todo el semestre y la vamos a abordar con especial énfasis 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 definen los requisitos que definen 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 (y estado) 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) 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. El modelo también puede estar formado por datos volátiles que se crean en tiempo real y desaparecen al cerrar el programa.
- **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 del 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. cambiamos 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 ≥ 17 .
- 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 UOCoban. Para crear este proyecto se ha utilizado el framework de desarrollo de videojuegos multiplataforma [libgdx](#), el cual se basa en Java y OpenGL. En este [vídeo](#) se explica los pasos que hemos seguido para crear el proyecto UOCoban con libgdx.

Si lo abres en IntelliJ, verás que la estructura difiere un poco de la que hemos usado habitualmente. Si te fijas, no aparece de primeras el directorio `src`. En cambio, aparecen dos directorios llamados `core` y `desktop`. Esto es así porque el *wizard* (configurador) de libgdx genera tantos directorios como subproyectos (salidas) hemos seleccionado. En nuestro caso sólo hemos seleccionado la opción `desktop`, obviando `Android`, `iOS` y `HTML`. La manera de trabajar con libgdx es que en el directorio `core` se programa la mayoría (el núcleo, `core`) del videojuego, y en cada una de las subcarpetas (`desktop`, `Android`, `iOS` y `HTML`), se programan las especificidades de cada entorno de ejecución. Así pues, si, por ejemplo, la clase `Player` se debe comportar igual para `Android`, `iOS` y `HTML`, pero diferente para `desktop`, entonces programamos dicha clase en el directorio `core` (la cual servirá para `Android`, `iOS` y `HTML`), mientras que en el directorio `desktop` creamos una clase `Person` especial para este entorno. Así pues, libgdx sobrescribe las clases de `core` con las indicadas en el entorno para la que vamos a generar el *output*. Fíjate que si abres `core` y `desktop`, ambos directorios se estructuran de igual manera. En ellas es donde aparece el directorio `src` que hemos usado a lo largo de la asignatura.



Importante: En el caso de esta Práctica 2, **SÓLO** trabajaremos dentro del directorio `src` de `core`.

A continuación explicamos los subdirectorios ubicados dentro del directorio `core` que son más importantes para realizar esta Práctica:

docs: contiene la documentación Javadoc del proyecto finalizado. Abre el fichero `index.html` en un navegador web para ver la documentación del programa



Importante: Consultar el Javadoc será vital para hacer esta Práctica 2, puesto que 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`, `views` y `controller`. Lo hemos organizado así porque, como hemos comentado, usaremos el patrón MVC.

A diferencia de los proyectos que hemos ido haciendo a lo largo de las PECs, los recursos del programa no los encontraremos en `src/main/resources`, sino en un directorio ubicado en la raíz del proyecto llamado `assets`. Será en `assets` donde 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.

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. La información mostrada en el diagrama de clases de la Práctica 1 es válida, pero es incompleta. Para hacer esta práctica **deberás tener en cuenta las especificaciones que se indiquen en este enunciado, en el Javadoc que se proporciona (directorio docs) y en los test. Recuerda que los test tienen prioridad en caso de contradicción.**

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/o 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 aún así no logras resolverlos, **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 .

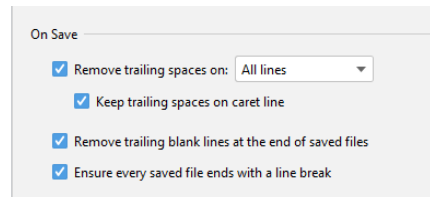
8. **Los test proporcionados no deben ser modificados.** Asimismo, cualquier práctica en la que se detecten trampas, p.ej. *hardcodear* código para que supere los test, será suspendida con un 0. En caso de discrepancia entre enunciado y test, quien manda es el test.


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.**

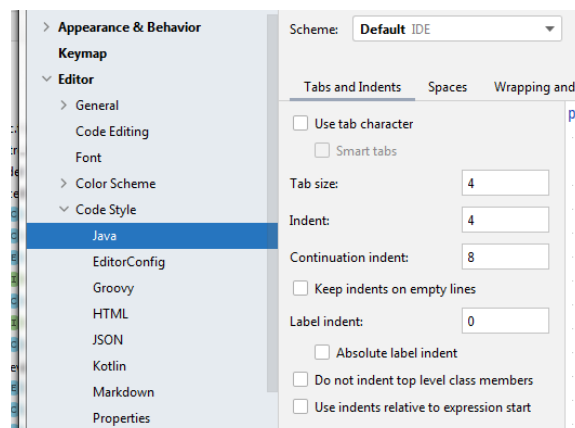
Antes de empezar

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

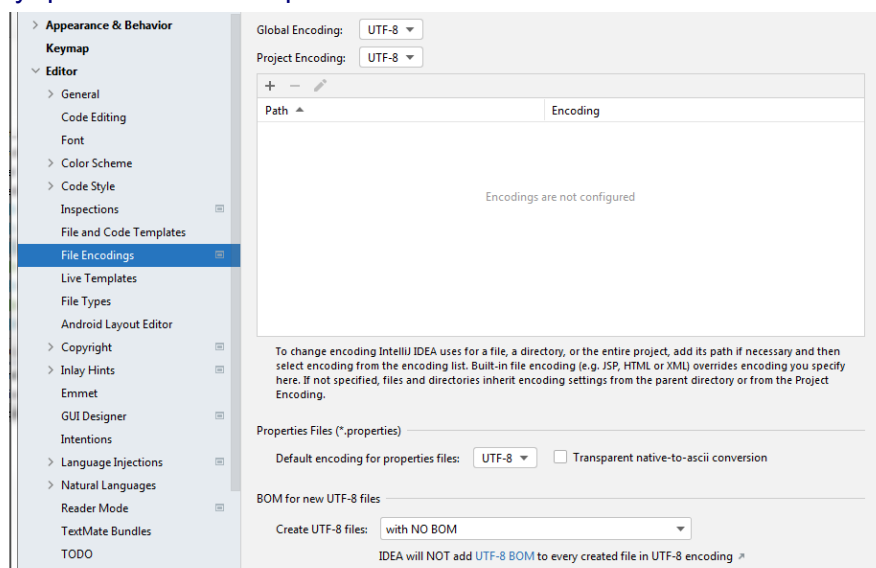
En la ventana escoge **Editor** → **General**. En la parte de la derecha, baja y marca todos los ítems del apartado **On Save**.



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"**.

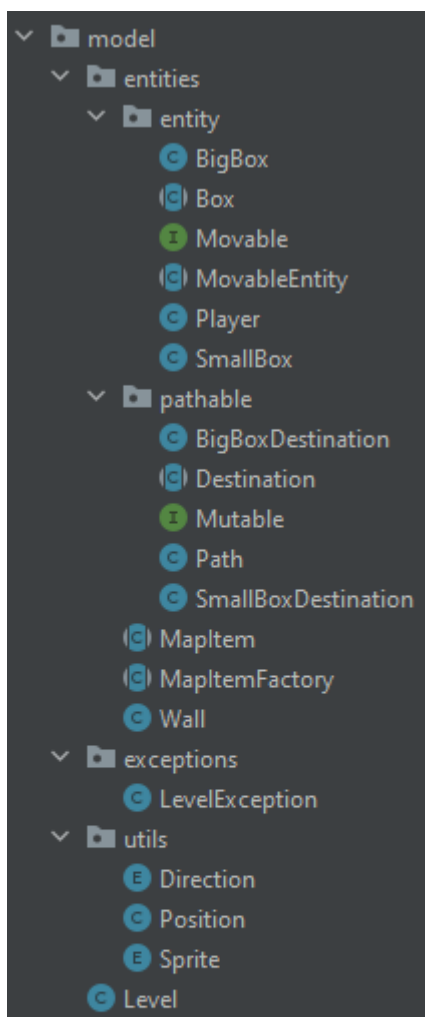


Modelo (8 puntos)

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



Para hacer esta Práctica, partiremos de la solución de la Práctica 1, pero con añadidos/modificaciones. Estos cambios los haremos notar, o bien en este PDF, o bien en la documentación generada con Javadoc, o bien en los test.



A continuación vamos a guiarte en la codificación del modelo, dándote información adicional que creemos necesaria, estableciendo un orden que te permita codificar el modelo siguiendo una cierta lógica y priorizando la codificación de los elementos más sencillos por delante de los más complejos (aunque no siempre será posible por la dependencia entre elementos).

Antes de nada, ten en cuenta que el paquete `model` (en `core`) se estructurará tal y como se muestra en la imagen de la izquierda, siendo `entities`, `exceptions` y `utils`, tres subpaquetes de `model`, donde `entities` se divide en 2 subpaquetes: `entity` y `pathable`.

Clase `MapItemFactory`

Esta clase es nueva y no aparece en el diagrama de clases de la solución de la Práctica 1, pero **te la damos ya codificada y no tienes que hacer nada con ella**. Las 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 instanciar un objeto de estas subclases/implementaciones concretas. No es parte de esta asignatura conocer este patrón. De hecho, 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/>.

Sugerencia de orden a seguir a la hora de codificar

A continuación ofrecemos un orden lógico a seguir a la hora de codificar el programa. Dicho orden tiene en cuenta la lógica del programa así como la dificultad de los elementos. Obviamente es una sugerencia y no es necesario seguirla de manera estricta.

Orden	Elementos	Orden	Elementos
1	LevelException	7	BigBoxDestination y SmallBoxDestination
2	Direction, Position y Sprite	8	MovableEntity
3	MapItem	9	Box
4	Wall y Path	10	BigBox y SmallBox
5	Mutable	11	Player
6	Destination	12	Level



Importante: Para poder ejecutar los test así como para codificar elementos más sencillos que dependen de otros más complejos (p.ej. `MovableEntity` tiene un atributo de tipo `Level`), hay que codificar primero el esqueleto de todos los elementos para que el programa compile, de lo contrario no se podrán ejecutar los test. **Ten presente el modelo de evaluación de esta Práctica 2, donde es necesario superar satisfactoriamente todos los test de tipo "sanity".**

Por "esqueleto" nos referimos a crear todos los elementos del programa con todos sus métodos codificados con un cuerpo/código mínimo que permita compilar el programa. Por ejemplo, si un método devuelve un `int`, podemos poner como cuerpo del programa simplemente:

```
return 1;
```

Más adelante se cambiará el cuerpo para que el método se comporte como es esperado.

Indicaciones

Clase `LevelException`

Codifica esta clase siguiendo las indicaciones del Javadoc.

Enum `Direction`

Codifica esta enumeración siguiendo las indicaciones del Javadoc. Asimismo ten en cuenta que el objetivo de esta enumeración es recopilar las direcciones/orientaciones en las que el jugador podrá desplazarse, i.e. derecha, abajo, izquierda y arriba.

Como verás en el Javadoc, cada dirección es “construida” a partir de tres parámetros: (1) `x`, (2) `y`, y (3) `keyCode`. Los valores `x` e `y` indican el *offset* en los que un elemento debe moverse si va en esa dirección. Así pues:

Valor	Valor para <code>x</code>	Valor para <code>y</code>	Valor para <code>keyCode</code>
RIGHT	1	0	22
DOWN	0	1	20
LEFT	-1	0	21
UP	0	-1	19

Clase `Position`

Codifica esta enumeración siguiendo las indicaciones del Javadoc. Asimismo, ten en cuenta también la especificación de los siguientes métodos:

- **`equals`**: adicionalmente a la especificación definida para el método `equals` de la clase `Object`, es necesario sobrescribir el método `equals` indicando que dos posiciones son iguales (i.e., devuelve `true`) si ambos valores para las coordenadas `x` e `y` coinciden. En caso contrario, devuelve `false`.
- **`hashCode`**: la documentación oficial del método `equals` en Java ([https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals\(java.lang.Object\)](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object))) establece la recomendación de sobrescribir el método `hashCode` cuando se sobrescribe el método `equals`. De esta manera, dos objetos que sean iguales deberían tener el mismo valor *hash* (`int`). Para ello, deberás obtener el valor obtenido al llamar al método estático `Objects.hash`, que recibirá como parámetros los valores `x` e `y`.
- **`toString`**: codifícala según las indicaciones del Javadoc.

Clase `Sprite`

Codifica esta enumeración siguiendo las indicaciones del Javadoc y los valores expuestos en el enunciado de la Práctica 1.

Fíjate que las rutas especifican la ubicación de los ficheros de imagen (formato `.png`) ubicados en la carpeta `assets/images` del proyecto.

Clase `MapItem`

Clase abstracta de la que heredarán todas las clases que modelan un elemento del mapa de juego (i.e., muros, caminos, cajas, destinos y jugador).

Fíjate que en la especificación la visibilidad del constructor está definida como `protected`, de manera que solamente las clases que hereden de `MapItem` tengan acceso al constructor. Dicho constructor recibe dos parámetros: (1) la `position` inicial del elemento; y (2) el `sprite` o representación gráfica del elemento.

Por otro lado, el método `isPathable` es definido como abstracto, ya que será cada elemento el responsable de determinar las características de la transitabilidad de un elemento movable sobre él mismo.

Clase `Wall`

Esta clase hereda de `MapItem`, y el valor de `sprite` es `Sprite.WALL`.

Por otro lado, `isPathable` siempre devuelve `false` (i.e., en ninguna circunstancia un muro puede ser transitable por un elemento movable).

Clase `Path`

Esta clase hereda de `MapItem`, y el valor de `sprite` es `Sprite.PATH`.

Por otro lado, `isPathable` siempre devuelve `true` (i.e., en cualquier circunstancia un camino puede ser transitable por un elemento movable).

Interfaz `Mutable`

Codifica esta enumeración siguiendo las indicaciones del Javadoc. Esta interfaz será implementada por cualquier elemento del mapa que pueda mutar su representación gráfica en un momento determinado del juego (i.e., `Destination`).

Clase `Destination`

Clase abstracta que hereda de `MapItem` y que implementa la interfaz `Mutable`. Para codificar esta clase, debes seguir las indicaciones del Javadoc.

Fíjate que `isPathable` devuelve `true` si el elemento destino está vacío (es decir, si no hay ninguna caja asignada). En caso contrario, devuelve `false`.

Clase `BigBoxDestination`

Esta clase hereda de `Destination`, y el valor de `sprite` es `Sprite.BIG_BOX_DESTINATION`.

Por otro lado, cuando una `BigBoxDestination` muta, el valor de `sprite` pasa a ser `Sprite.BIG_BOX_ON_DESTINATION`.

Clase `SmallBoxDestination`

Esta clase hereda de `Destination`, y el valor de `sprite` es `Sprite.SMALL_BOX_DESTINATION`.

Por otro lado, cuando una `SmallBoxDestination` muta, el valor de `sprite` pasa a ser `Sprite.SMALL_BOX_ON_DESTINATION`.

Clase `MovableEntity`

Esta clase abstracta hereda de `MapItem`. De esta clase heredarán todos los elementos movibles del juego (i.e., `Player` y `Box`).

Para codificar esta clase debes seguir las indicaciones del Javadoc (fíjate que `MovableEntity` especifica el método abstracto `move`).

Clase `Box`

Esta clase abstracta hereda de `MovableEntity`. De esta clase heredarán todos los tipos de cajas del juego (i.e., `BigBox` y `SmallBox`). Ten en cuenta que el constructor parametrizado inicializa el atributo `delivered` con el valor `false`.

Por otro lado, `isPathable` siempre devuelve `false` (i.e., en ninguna circunstancia una caja puede ser transitable por un elemento movable).

El resto de indicaciones (incluyendo los detalles de la especificación del método `move`) deben consultarse en el Javadoc.

Clase `BigBox`

Esta clase hereda de `Box`, y el valor de `sprite` es `Sprite.BIG_BOX`.

Clase `SmallBox`

Esta clase hereda de `Box`, y el valor de `sprite` es `Sprite.SMALL_BOX`.

Clase `Player`

Esta clase hereda de `MovableEntity`, y el valor de `sprite` es `Sprite.PLAYER`.

El resto de indicaciones (incluyendo los detalles de la especificación del método `move`) deben consultarse en el Javadoc.

Clase `Level`

Para esta clase te proporcionamos su esqueleto y algún método ya codificado que no debes modificar. El resto de métodos (i.e., aquellos que no te damos implementados) tienen el comentario `//TODO`.



Importante: 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`.

Los métodos sin codificar son: `setWidth`, `setHeight`, `setRemainingMovements`, `decRemainingMovements` y `hasWon`. Estos métodos deben codificarse considerando la especificación en comentarios Javadoc.

Por otro lado, la clase `Level` contiene una serie de métodos asociados con el control de si un nivel ha alcanzado la condición de *deadlock*. Esta condición se establece cuando la disposición de los elementos del mapa del juego hace imposible su resolución a causa del bloqueo de cajas y, por tanto, de la incapacidad de realizar ningún conjunto de movimientos que permita entregar alguna de las cajas en una destinación. Estos métodos son:

- `isPathableOrMovable`: determina si, para una determinada posición (expresada por parámetros), el elemento del mapa ubicado en esa posición es transitable (i.e., *pathable*) o bien movable (i.e., *movable*). En los comentarios del Javadoc encontrarás más información.
- `isDeadlocked`: este método comprueba si cualquiera de las cajas que forman parte del nivel actual está en situación de *deadlock*. En caso de que cualquiera de las cajas que componen el mapa de juego cumpla la condición de *deadlock*, este

método devuelve `true`; en caso contrario, devuelve `false`. Fíjate que este método está sobrecargado con una versión parametrizada, en la que recibimos por parámetro un objeto `Box` sobre el que evaluamos la condición de *deadlock*. En los comentarios del Javadoc encontrarás más información acerca de las condiciones que este método debe considerar como *deadlock*.

Fíjate que, aunque los comentarios Javadoc establecen claramente las condiciones exactas que consideraremos como *deadlock* en esta práctica, no se dan detalles a bajo nivel sobre las comprobaciones a realizar, ya que se espera que seas capaz de evaluar e implementar el algoritmo con cierta autonomía.



Pista: Utiliza el método `isPathableOrMovable` en la codificación del método `isDeadlocked` para facilitar su implementación, así como su legibilidad.

Controlador (1.5 puntos)

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). Verás que, de la misma forma que la clase `Level`, esta clase contiene el esqueleto completo (i.e., especificación de métodos y atributos) y la implementación de algunos de sus métodos. El resto de métodos tienen el comentario `//TODO` y los debes codificar siguiendo las especificaciones indicadas en el Javadoc.

Vistas (0.5 puntos)

Las vistas son las “pantallas” con las que interactúa el usuario. En este caso, tenemos una manera de interactuar, es decir, el juego en modo gráfico. Con el proyecto ya te damos las vistas/pantallas del programa hechas, puesto que están realizadas con el framework `libgdx` que no es objeto de estudio de esta asignatura. Estas vistas se encuentran en el package `edu.uoc.uocoban.view`.

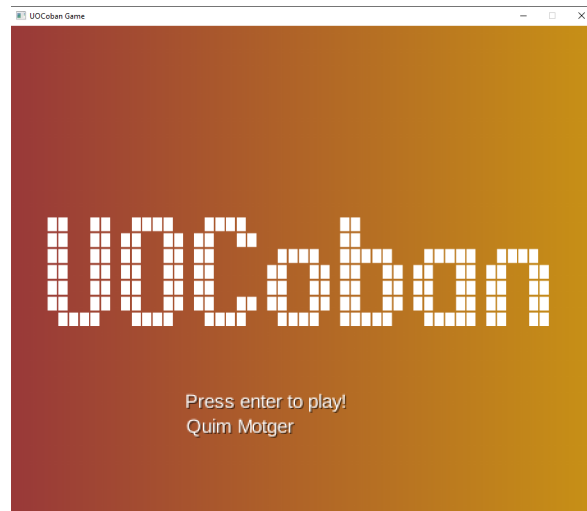
Para ejecutar esta vista, verás que la ventana de Gradle de IntelliJ muestra dentro de `Tasks > other` una tarea llamada `run`. Si ejecutas esta tarea, se ejecutará el juego en modo gráfico. La manera de jugar es usando las flechas del teclado.

Con el objetivo de conocer algo más a fondo los elementos que componen el patrón MVC, deberás editar las vistas para añadir algunos elementos adicionales que, actualmente, están ausentes. Estos elementos son:

- **WelcomeScreen:** esta vista implementa la ventana de inicio del juego, previa a la carga de cualquier nivel. En la versión que te damos, esta vista incluye una imagen

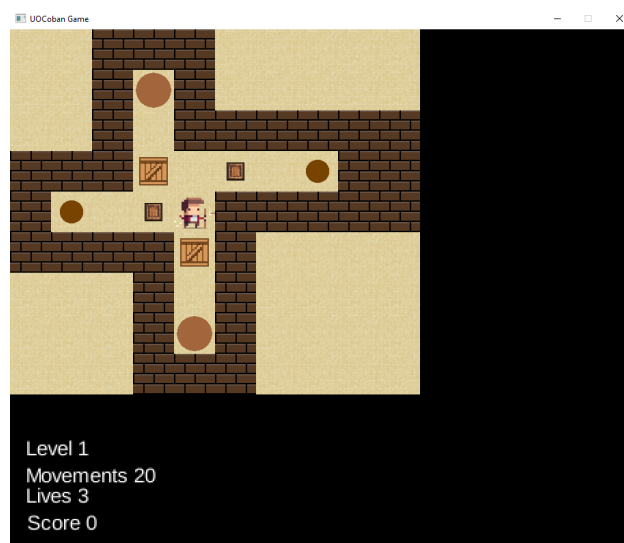
con el nombre del juego UOCoban, y un mensaje con el texto "Press enter to play!".

En este ejercicio te pedimos que modifiques el código de la vista (sustituyendo el comentario `//TODO`) de manera que, debajo del mensaje anterior, añadas tu nombre completo, tal como se muestra en la siguiente imagen.



- **GameScreen:** esta vista implementa la visualización de un nivel en curso. En la versión que te damos, esta vista incluye, por un lado, el mapa completo del juego, con todos sus elementos (personaje, cajas, destinos, etc.). Por otro lado, incluye información acerca del nivel, pero está incompleta (solamente mostramos el nivel actual de juego).

En este ejercicio, te pedimos que modifiques esta vista para añadir, tal y como se muestra en la siguiente figura, la información ausente. Concretamente: (1) número de movimientos restantes del nivel; (2) número de vidas restantes del juego; y (3) puntuación actual del juego.



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 realizada con libgdx puedes ir a la tarea de Gradle llamada `other → dist`, la cuál creará un directorio `build` en cada “proyecto”, en nuestro caso `core` y `desktop`. Si miras dentro, verás que hay un subdirectorio llamado `libs → dist`, en él hay el correspondiente `.jar`. Este es un *fat jar*, es decir, un fichero `.jar` que, además de las clases de nuestro programa, contiene también todas las clases de todas las librerías de las que depende. Así pues, es un fichero más grande (de ahí el uso del adjetivo *fat*) de lo que sería un `.jar` generado de manera normal, puesto que contiene también los ficheros e imágenes. Puedes ejecutarlo haciendo doble click o usando el comando `java -jar desktop-1.0.jar` en un terminal (puedes cambiarle el nombre al fichero). Este fichero `.jar` funcionará en aquellos ordenadores que tengan tanto JDK como JRE. Si tienes curiosidad, puedes descomprimirlo con WinZip (o similar) para ver su interior.

Quizás estés pensando: *¿qué sucede si en el ordenador en que se quiere ejecutar el `.jar` no hay ni JDK ni JRE?* Pues, o bien lo instalas, o bien usas `jlink`. 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. Para usar `jlink` hay que configurar la tarea `jar` de `build.gradle`. Esto queda fuera del alcance de la asignatura.

Cabe destacar que `jlink` es un comando propio de JDK y, por lo tanto, se puede ejecutar desde línea de comandos sin necesidad de usar Gradle (y el plugin correspondiente): <https://www.devdungeon.com/content/how-create-java-runtime-images-jlink>.

¿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>.

De todas maneras, hoy en día se usan aplicaciones como Docker para distribuir programas.

Criterios de evaluación

En este apartado indicamos cuáles son los criterios de evaluación de esta práctica:

Requisito imprescindible para evaluar la Práctica 2

testSanity	Ejecución	Gradle → verification → testSanity
	Descripción	Este test suite asegura que la parte crítica del esqueleto del programa es respetada. Todos los test de testSanity deben ser pasados satisfactoriamente para que la práctica sea evaluada. Si alguno de estos test falla, entonces la nota que obtendrás en la Práctica 2 será un 1 independientemente del resto de test.



Importante: Solo si se superan todos los test que conforman testSanity se evaluarán, de manera independiente, los test suite que se indican a continuación.

Evaluación Modelo (8 puntos)

initLevel	Ejecución	Gradle → verification → initLevel
	Descripción	<p>Este test suite comprueba que los métodos asociados a la inicialización de un nivel (así como los <i>getters</i> y <i>setters</i> de cada clase) son funcionalmente correctos.</p> <p>La nota se calculará a partir de la siguiente fórmula:</p> $\frac{(\#test_init_level_pasados}{\#test_init_level)} * 4.0$
playLevel	Ejecución	Gradle → verification → playLevel
	Descripción	<p>Este test suite comprueba que los métodos asociados a la evolución de un nivel son funcionalmente correctos.</p> <p>La nota se calculará a partir de la siguiente fórmula:</p> $\frac{(\#test_play_level_pasados}{\#test_play_level)} * 3.0$
deadlock	Ejecución	Gradle → verification → deadlock
	Descripción	<p>Este test case comprueba que los métodos asociados a la comprobación del estado de <i>deadlock</i> de un nivel son funcionalmente correctos.</p> <p>La nota se calculará a partir de la siguiente fórmula:</p> $\frac{(\#test_deadlock_pasados}{\#test_deadlock)}$



Importante: El profesor se reserva la posibilidad de penalizar con hasta **-0,5 puntos** en función de la calidad y legibilidad del algoritmo implementado para controlar el estado de *deadlock*.

Evaluación Controlador (1.5 puntos)

GameTest	Ejecución	<code>Gradle → verification → initLevel</code>
	Descripción	<p>Este test suite comprueba que los métodos asociados a la lógica del controlador son funcionalmente correctos.</p> <p>La nota se calculará a partir de la siguiente fórmula:</p> $(\#test_game_pasados / \#test_game) * 1.5$

Evaluación Vista (0.5 puntos)

Elemento	Puntuación
WelcomeScreen	0.2 puntos
GameScreen	0.3 puntos

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 UOCoban 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 **5 de julio de 2023** antes de las 23:59. Cualquier Práctica entregada más tarde será considerada como no presentada.