

PRÁCTICA 2

Codificación

UOC

Información relevante:

- Fecha límite de entrega: 28 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
Modelo	9
Sugerencia de orden a seguir a la hora de codificar	10
Indicaciones	10
Ball (clase)	10
Brick (clase)	11
BrickBasic, BrickAdvanced y BrickUnbreakable (clases)	11
Entity (clase)	11
Level (clase)	11
LevelException (clase)	11
Movable (interfaz)	11
Paddle (clase)	11
XDirection e YDirection (enumeraciones)	12
Controlador	12
Vistas	14
Corolario	15
Evaluación	17
Formato y fecha de entrega	18

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:

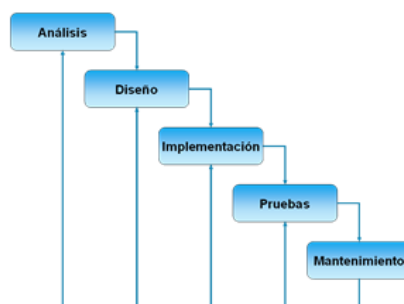
- Codificar un programa basado en un patrón de arquitectura como es MVC.
- Utilizar diferentes fuentes de información para desarrollar un programa..
- Usar ficheros de test en JUnit para determinar que un programa es correcto.
- Usar con cierta soltura un entorno de desarrollo integrado (IDE) como IntelliJ.

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 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.

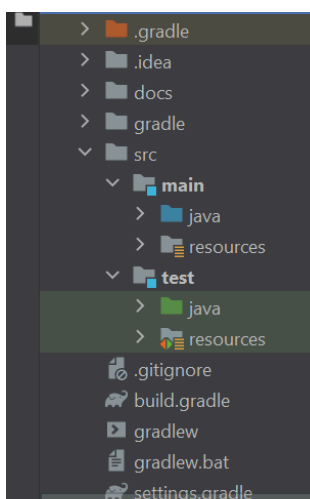
Entorno

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 UOCanoid. 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 4 subpaquetes), `view` y `controller`. Lo hemos organizado así porque, como hemos comentado, usaremos el patrón MVC.

En `src/main/resources` encontrarás los estilos 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: contiene toda la configuración necesaria de Gradle. En él hemos definido tareas específicas para esta Práctica con la finalidad de ayudarte durante su realización.

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 puede ser un poco diferente al diseño usado para esta Práctica 2. 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 aun 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.

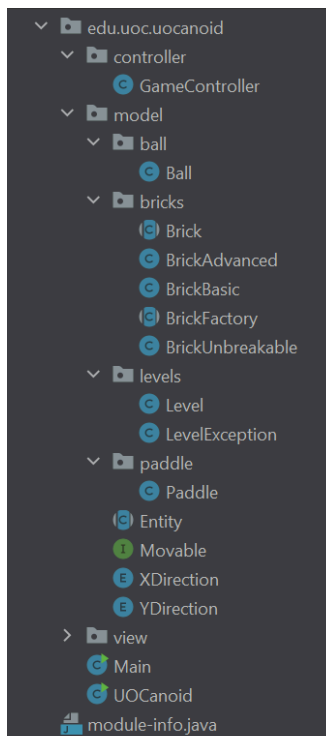
9. **La fecha límite indicada en este enunciado es inaplazable.** Puedes hacer diversas entregas durante el período de realización de la Práctica. El Equipo Docente corregirá la última entrega. Asegúrate de que entregas los ficheros correctos. **Una vez finalizada la fecha límite, no se aceptarán más entregas.**

Modelo

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



Con este enunciado te proporcionamos el diagrama de clases UML del modelo y controlador que vamos a utilizar. En términos generales, es muy similar a la solución de la Práctica 1.



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` se estructurará tal y como se muestra en la imagen de la izquierda, siendo `ball`, `bricks`, `levels` y `paddle`, cuatro subpaquetes de `model`.

Clase `BrickFactory`

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**. 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 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	Movable	7	Brick
2	LevelException	8	BrickBasic
3	Entity	9	BrickAdvanced
4	XDirection	10	BrickUnbreakable
5	YDirection	11	Paddle
6	Level	12	Ball



Importante: Para poder ejecutar los test así como para codificar elementos más sencillos que dependen de otros más complejos (p.ej. `Level` tiene un atributo de tipo `Ball`), 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.

Para codificar todas las clases, interfaces y enumeraciones sigue las indicaciones proporcionadas en el Javadoc. No obstante, ten presente los detalles que se indican en el siguiente apartado (están enumerados por orden alfabético) que pueden ser de ayuda.

Indicaciones

Ball (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc y el diagrama de clases. Ten en cuenta que el método `move` (procedente de la interfaz `Movable`) debe actualizar el valor de

los atributos `x` e `y` sumando o restando una cantidad igual a `speed` según sea el valor de los atributos `xDirection` e `yDirection`.

Brick (clase)

Clase abstracta que modela un ladrillo genérico. Codifica esta clase siguiendo las indicaciones del Javadoc y el diagrama de clases. Ten en cuenta que el atributo `currentHits` debe ser inicializado a 0.

BrickBasic, BrickAdvanced y BrickUnbreakable (clases)

Codifica estas clases siguiendo las indicaciones del Javadoc y el diagrama de clases. Estas clases modelan sendas representaciones específicas de un ladrillo.

Entity (clase)

Clase abstracta que modela un elemento del juego. Codifica esta clase siguiendo las indicaciones del Javadoc y el diagrama de clases.

Level (clase)

Para esta clase te proporcionamos su esqueleto y casi todos los métodos ya codificados que no debes modificar. El resto de métodos tienen el comentario `//TODO` y los debes codificar siguiendo las especificaciones indicadas en el Javadoc.

Ten presente que el método `isCompleted` debe devolver `true` si no quedan ladrillos en el nivel o si los que quedan son de tipo `BrickUnbreakable`.



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`.

LevelException (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc y el diagrama de clases.

Movable (interfaz)

Codifica esta interfaz siguiendo las indicaciones del Javadoc y el diagrama de clases.

Paddle (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc y el diagrama de clases. Ten en cuenta que el método `move` (procedente de la interfaz `Movable`) debe actualizar el valor del atributo `x` sumando o restando una cantidad igual a `speed` según sea el valor del atributo `direction`.

XDirection e YDirection (enumeraciones)

Codifica estas enumeraciones que son prácticamente idénticas siguiendo las indicaciones del Javadoc y el diagrama de clases. Asimismo ten en cuenta que el objetivo de estas enumeraciones es recopilar las direcciones/orientaciones en las que los elementos movibles (i.e. pelota y pala) podrán desplazarse en los respectivos ejes X (i.e. derecha e izquierda) e Y (i.e. arriba y abajo).

Como verás en el Javadoc y el diagrama de clases, cada dirección es “construida” a partir de un parámetro llamado `orientation`. Este valor permite saber en qué orientación hay que mover un elemento según una dirección dada. Así pues:

Enum	Valor	Valor para <code>orientation</code>
XDirection	LEFT	-1
	RIGHT	+1
YDirection	UP	-1
	DOWN	+1

El método `next` de ambas enumeraciones devuelve el siguiente valor de la enumeración. Así pues, si se invoca `LEFT.next()` entonces el valor a devolver debe ser `RIGHT`. Si se invoca `RIGHT.next()` el valor a devolver es `LEFT`. Lo mismo con los valores de `YDirection`.

El método `getValueByOrientation` de ambas enumeraciones devuelve el valor de la enumeración que tiene asociado el valor de `orientation` pasado como parámetro. Así pues, si se invoca `XDirection.getValueByOrientation(1)` debe devolver `RIGHT`. Si el valor pasado por parámetro no tiene coincidencia, entonces devuelve `null`.

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 que el controlador haga una petición al modelo.

En el paquete `controller` del proyecto verás una clase llamada `GameController`. Ésta es la clase controladora del juego (un programa puede tener varias clases controladoras). Ya te damos casi todos los métodos codificados (que no debes modificar). El resto de métodos tienen el comentario `//TODO` y los debes codificar siguiendo las especificaciones indicadas en el Javadoc y en este enunciado.

En esta clase queremos dar unas indicaciones adicionales para algunos métodos que debes codificar:

checkCollisionBricks

Para cada ladrillo que esté aún sin destruir, debe comprobar si la pelota está dentro del área del ladrillo. Ten en cuenta que las posiciones `x` e `y` de las entidades hacen referencia a la coordenada de la esquina superior izquierda de dicho elemento. Así pues, la pelota estará dentro del rango de un ladrillo si la `x` de la pelota es menor o igual al extremo derecho del ladrillo (`x + width`) y a su vez la esquina superior derecha de la pelota (`x + width`) es igual o superior a la `x` del ladrillo. Además de lo anterior se debe cumplir unas condiciones similares para el eje `Y`. En este caso, ten presente que el eje `Y` se incrementa hacia abajo.

Si la pelota está dentro del rango de un ladrillo, entonces la pelota cambiará su dirección en el eje `Y`. No obstante, cabe determinar si también debe cambiar su dirección en el eje `X`. Así pues, si la pelota está dentro del rango de un ladrillo, este método debe comprobar si la pelota ha golpeado el ladrillo por la izquierda o por la derecha (en el Javadoc se indica cómo determinarlo). En caso de haberlo hecho, cambiará la dirección de la pelota en el eje `X`.

Una vez se ha determinado que la pelota ha golpeado un ladrillo, entonces se debe incrementar el número de golpes recibidos por ese ladrillo y actualizar la puntuación del juego si como consecuencia ha sido destruido. Al destruir un ladrillo, este método debe eliminarlo de la lista de ladrillos del nivel (te recomendamos echar un vistazo a cómo funcionan los `Iterator` en Java).

checkCollisionScene

Comprueba si la pelota excede el tamaño de la pantalla del juego. Si excede por la izquierda (`x` de la pelota inferior o igual a 0) o por la derecha (`x + width` de la pelota igual o superior a `WIDTH`), entonces se debe cambiar la dirección de la pelota en el eje `X`. Si excede por arriba (`y` de la pelota igual o inferior a 0), entonces se debe cambiar la dirección de la pelota en el eje `Y`.

checkCollisionBottom

Determina si la `y` de la pelota excede (es estrictamente superior) a la `y` de la pala. Si lo es, resta una vida y restaura el juego.

checkCollisionPaddle

Comprueba si la pelota está dentro del rango/área/zona de la pala. Esta comprobación es muy parecida a la realizada en `checkCollisionBricks`. Si la pelota está dentro de la zona de la pala, entonces debe cambiar la dirección de la pelota en el eje `Y`. En el caso del eje `X`, solo cambiará si la pelota ha colisionado por las “esquinas” de la pala, no en la zona central. Esto significa que la `x` de la pelota es igual o inferior a la `x` de la pala (colisión por la

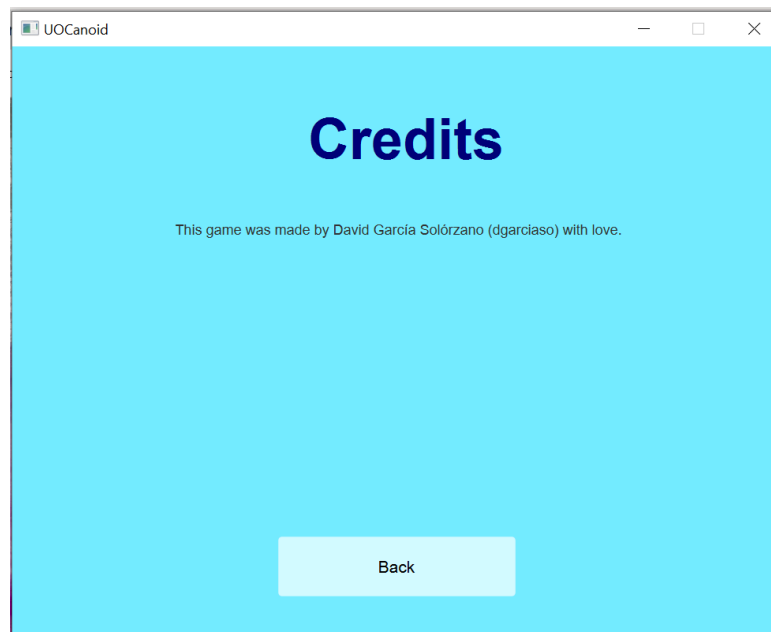
izquierda), o que la x de la pelota es igual o mayor a la esquina superior derecha/izquierda de la pala ($x + width$) menos el ancho de la pelota.

movePaddle

Mueve la pala a una nueva posición solo si dicha posición que tendrá la pala hace que alguna parte de la pala no salga de la escena. Es decir, que la nueva x de la pala no sea inferior a 0, ni la nueva x más el `width` de la pala sea superior al atributo `WIDTH` del juego.

Vistas

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”. Decimos “hechas” porque te pedimos que añadas un texto con tu nombre y login en la pantalla `credits.fxml` y un botón que permita volver al menú principal (i.e. `main.fxml`).



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. Cuando se abre un fichero `.fxml` en IntelliJ, éste muestra dos pestañas, una con el código FXML y otra pestaña "Scene Builder" que integra Scene Builder dentro del IDE.

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 dentro de la configuración del plugin `application` usa como `main`, el que tiene `UOCanoid`, mientras que la tarea `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 indicar 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`. En el apartado 4.3 de la Guía de Java damos una pincelada muy breve al tema de los módulos introducidos por JDK 9.

Para crear un fichero `.jar` que se ejecute en una máquina que tenga instalada JRE, debes descomentar la tarea `jar` que encontrarás dentro de `build.gradle`. Esta tarea está configurada para crear 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. Una vez descomentada la tarea y actualizadas las tareas Gradle (recuerda darle al botón refrescar que aparece en el fichero `build.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 `UOCanoid-1.0-SNAPSHOT.jar` (contiene todo: `.class` y recursos) y ejecútalo donde quieras (asegúrate que en el ordenador que utilices esté, como mínimo, la versión 17 de JRE). Puedes ejecutarlo haciendo doble click o usando el comando `java -jar UOCanoid-1.0-SNAPSHOT.jar` en un terminal.

Quizás estés pensando: *¿qué sucede si en el ordenador en que se ejecuta el `.jar` no hay 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` debes comentar, en `build.gradle`, la tarea `jar` que genera el *fat jar*. A continuación, descomentar la tarea `jlink` que encontrarás en `build.gradle`. Después, sólo tienes que hacer doble click en la tarea de Gradle llamada `jlink` que encontrarás dentro del grupo `build`. El resultado se creará en la carpeta `build/image`. Para ejecutar la aplicación debes ir a `image/bin` y ejecutar el fichero `UOCanoid`, no sin antes copiar el directorio `levels` que hay en `resources` dentro del directorio `bin` (sinceramente, no sabemos por qué no funciona dejándolo en `resources`). A veces hay problemas para que la aplicación generada con `jlink` lea correctamente los ficheros añadidos en `resources`, así que si no os funciona, no os frustréis.

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.


Evaluación

Esta Práctica se evalúa a partir de la superación de test.

Requisito imprescindible para evaluar la práctica

Tipo de test	Comentarios
40 sanity	<p>Estos test aseguran que la parte crítica del esqueleto del programa es respetada. Para probarlos haz:</p> <pre>Gradle → verification → testSanity</pre> <p>Estos test deben ser pasados satisfactoriamente en su totalidad para que la práctica sea evaluada. Así pues, si alguno de estos test falla, entonces la nota que obtendrás en la Práctica 2 será un 1 independientemente del resto de test (i.e. "basic" y "advanced") y de la parte de la vista (i.e. interfaz gráfica).</p>

Tipos de test y peso en la evaluación

Tipo de test	Peso	Comentarios
86 basic	70%	<p>Estos test comprueban que los métodos básicos son funcionalmente correctos. Para probarlos haz:</p> <pre>Gradle → verification → testBasic</pre> <p>La nota se calculará a partir de la siguiente fórmula:</p> $(\#test_basic_pasados / \#test_basic) * 70\%$
7 advanced	20%	<p>Estos test comprueban que los métodos avanzados son funcionalmente correctos. Para probarlos haz:</p> <pre>Gradle → verification → testAdvanced</pre> <p>La nota se calculará a partir de la siguiente fórmula:</p> $(\#test_advanced_pasados / \#test_advanced) * 20\%$
<p> Gradle → verification → testSanityBasic ejecuta todos los test de tipo "sanity" y "basic". Si se superan los 126 test, la nota de la Práctica es, como mínimo, un 7.</p> <p>Gradle → verification → testAll ejecuta todos los test. Si se superan los 133 test, la nota de la Práctica es, como mínimo, un 9.</p>		

Evaluación de la vista (interfaz gráfica) = 1 punto

Añadir el texto solicitado en la pantalla de créditos = 0.25 pts.

Añadir el botón de "back" en la pantalla de créditos = 0.25 pts.

Funcionamiento correcto del botón "back" de la pantalla de créditos = 0.5 pts.

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 `UOCanoid` 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 **28 de enero de 2024** antes de las 23:59. Cualquier Práctica entregada más tarde será considerada como no presentada.