PRÁCTICA 2

Codificación

Universitat Oberta de Catalunya

Información relevante:

- Fecha límite de entrega: 29 de enero.
- Peso en la nota final de Prácticas: 70%.

Contenido

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:

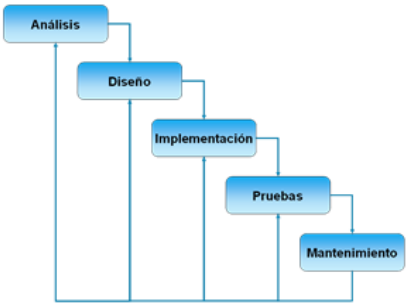
- Codificar un programa basado en un patrón de arquitectura como es MVC.
- Utilizar un framework de desarrollo de videojuegos para Java..
- Usar ficheros de test en JUnit para determinar que un programa es correcto.
- Usar con cierta soltura un entorno de desarrollo integrado (IDE) como IntelliJ.

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 `PACman`. 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 `PACman` 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 `Person` 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 sobreescribe 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. Es 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 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 .

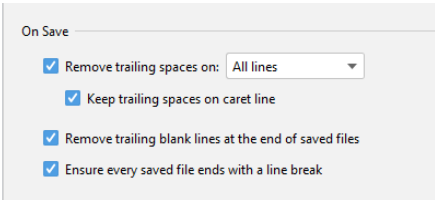
8. **Los test proporcionados no deben ser modificados**. Asimismo, cualquier práctica en la que se detecten trampas, p.ej. *hardcodear* código para que supere los test, será suspendida con un 0. En caso de discrepancia entre enunciado y test, quien manda es el test.

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 nuevas entregas**.

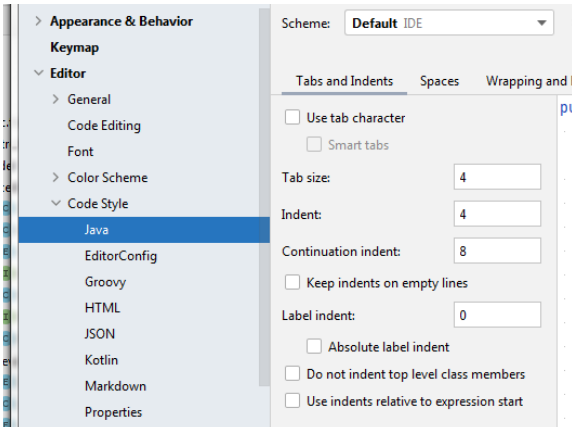
Antes de empezar

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

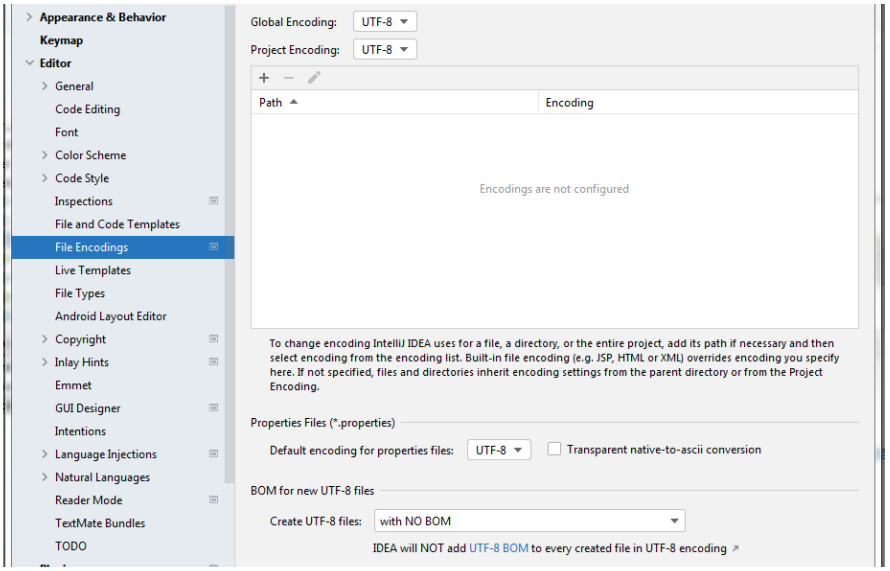
En la ventana escoge `Editor → General`. En la parte de la derecha, baja y marca todos los ítems del apartado `On Save`.



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 `V` 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"`.

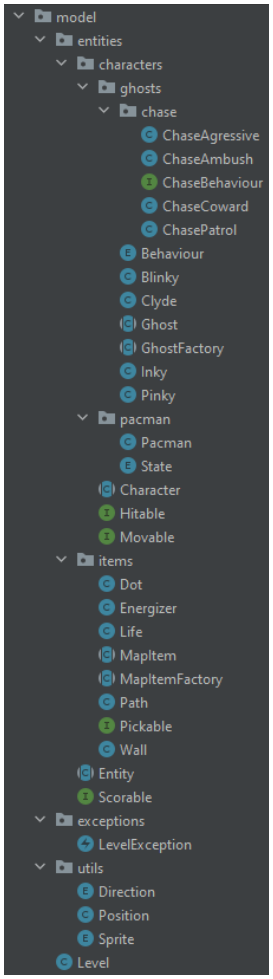


Modelo

En la Práctica 1 se te pidió que hicieras el diagrama de clases UML del modelo (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: `characters` e `items`. A su vez `characters` incluye los paquetes `ghosts` y `pacman`. Finalmente, el paquete `ghosts` tiene el paquete `chase`.

Clases `MapItemFactory` y `GhostFactory`

Estas dos clases son nuevas y no aparecen en el diagrama de clases de la solución de la Práctica 1, pero **te las damos ya codificadas y no tienes que hacer nada con ellas**. 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=lLvYAzXO7Ek.

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	8	ChaseBehaviour
2	Direction, Position y Sprite	9	ChaseAggressive y ChaseAmbush
3	Entity y Scorable	10	ChaseCoward y ChasePatrol
4	Pickable y MapItem	11	Behaviour y State
5	Path, Wall, Life, Dot y Energizer	12	Ghost
6	Hitable y Movable	13	Blinky, Pinky, Inky y Clyde
7	Character	14	Pacman
15	Level		

Importante: Para poder ejecutar los test así como para codificar elementos más sencillos que dependen de otros más complejos (p.ej. `Character` 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.

Orientación acerca de la dificultad de los elementos a codificar

La siguiente tabla muestra una orientación sobre la dificultad de los diferentes elementos que componen el modelo del programa. Dentro de cada nivel de dificultad, los distintos elementos están ordenados, aproximadamente, de menor a mayor dificultad. Así pues, la interfaz `Scorable` sería más sencilla que la clase `Life`.

Fácil	Media
Scorable (package entities.characters.pacman)	Direction (package utils)
Movable (package entities.characters)	Sprite (package utils)
Hitable (package entities.characters)	Position (package utils)
Pickable (entities.items)	State (entities.characters.pacman)
ChaseBehaviour (entities.characters.ghosts.chase)	Behaviour (entities.characters.ghosts)
LevelException (exceptions)	Entity (entities)
MapItem (entities.items)	Character (entities.characters)
Path (entities.items)	Blinky (entities.characters.ghosts)
Wall (entities.items)	Pinky (entities.characters.ghosts)
Life (entities.items)	Inky (entities.characters.ghosts)
Dot (entities.items)	Clyde (entities.characters.ghosts)
Energizer (entities.items)	ChaseCoward (entities.characters.ghosts.chase)
ChaseAggressive (entities.characters.ghosts.chase)	ChasePatrol (entities.characters.ghosts.chase)
ChaseAmbush (entities.characters.ghosts.chase)	Level (model, va directamente en la raíz de model)
Alta	
Pacman (entities.characters.pacman)	
Ghost (entities.characters.ghosts)	

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

Blinky, Pinky, Inky y Clyde (clases)

Ten en cuenta que cada uno de ellas instancia un objeto diferente para el atributo `chaseBehaviour`.

ChaseAggressive (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc. Este comportamiento es el asociado a los fantasmas de tipo `Blinky`. Su posición objetivo es la posición en la que está Pacman. Por este motivo es el tipo de persecución más agresivo.

ChaseAmbush (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc. Este comportamiento es el asociado a los fantasmas de tipo `Pinky`. Intenta adelantarse a los movimientos de Pacman para tenderle una emboscada. Para ello su posición objetivo es 4 posiciones hacia adelante respecto a la posición actual de Pacman y en la dirección en la que está mirando Pacman.

Recuerda que el enum `Direction` almacena para cada valor su *offset*. Por ejemplo, para `UP`, éste indica que el ítem debe moverse `-1` en el eje Y, pero `0` en el eje X.

ChaseBehaviour (interfaz)

Codifica esta interfaz siguiendo las indicaciones del Javadoc. Gracias a esta interfaz y las clases que la implementan (i.e. `ChaseXXX`), estamos aplicando el patrón de diseño llamado Strategy. Dicho patrón se sustenta en el principio de Polimorfismo.

ChaseCoward (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc. Este comportamiento es el asociado a los fantasmas de tipo `Clyde`. Primero calcula la distancia euclídea que hay entre su posición y la de Pacman. Si es menor a 8, su posición objetivo es la misma posición que cuando está en estado `SCATTER`. En caso contrario, su posición objetivo es la posición de Pacman. Con este comportamiento a la hora de perseguir, da la sensación que este tipo de fantasma se comporta de forma cobarde.

ChasePatrol (clase)

Codifica esta clase siguiendo las indicaciones del Javadoc. Este comportamiento es el asociado a los fantasmas de tipo `Inky`. Para determinar su posición objetivo, los fantasmas de tipo `Inky` usan la posición y orientación de Pacman así como la posición del primer fantasma `Blinky` añadido al nivel. Primero calcula, teniendo en cuenta la dirección en la que está orientado Pacman, la posición que está dos posiciones delante de la posición actual de Pacman. Una vez determinada esta nueva posición (llamémosla `targetBlinky`), calcula un vector que va desde la posición del primer fantasma de tipo `Blinky` añadido al nivel hasta la posición `targetBlinky`. Con el vector calculado, lo dobla (i.e. lo multiplica por 2). El resultado es la posición objetivo a la que debe ir `Inky`. Con esta manera de moverse, los fantasmas `Inky` parecen que estén patrullando. En caso de que no haya ningún fantasma de tipo `Blinky` en el nivel, entonces la posición objetivo será `targetBlinky`.

Direction (enumeración)

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 `x`	Valor para `y`	Valor para `keyCode`
RIGHT	1	0	22
DOWN	0	1	20
LEFT	-1	0	21
UP	0	-1	19

Dot (clase)

El valor por defecto del atributo `pathable` es `true` y de sprite es `Sprite.DOT`.

Energizer (clase)

El valor por defecto del atributo `pathable` es `true` y de sprite es `Sprite.ENERGIZER`.

Ghost (clase)

No debemos poder instanciar objetos cuyo tipo dinámico sea `Ghost`. Además ten en cuenta que cuando se resetea un fantasma, éste vuelve a estar vivo en su posición inicial (i.e. la indicada en el fichero de configuración del nivel), con un comportamiento igual a `INACTIVE` y con dirección/orientación `UP`.

Level (clase)

Para esta clase 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.

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`.

Life (clase)

El valor por defecto para el atributo `pathable` es `true` y para sprite es `Sprite.LIFE`.

MapItem (clase)

Esta clase hereda de `Entity` y tiene un constructor. No debemos poder instanciar objetos cuyo tipo dinámico sea `MapItem`.

Pacman (clase)

Ten en cuenta que cuando se reseteaPacman, éste vuelve a estar vivo en su posición inicial (i.e. la indicada en el fichero de configuración del nivel), con un estado igual a `INVINCIBLE` y con dirección/orientación `UP`.

Path (clase)

El valor por defecto para el atributo `pathable` es `true` y para sprite es `Sprite.PATH`.

Position (clase)

Codifica esta clase según las indicaciones del Javadoc. También ten en cuenta:

- **`equals`**: la sobrescritura de este método debe devolver `true` cuando las dos posiciones 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 `Position`), 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 devolver el valor del método estático `hash` de la clase `Objects`, al cual debes pasarle como parámetros los valores de `x` e `y`, en este orden (para que no falle el test).
- **`add`**: devuelve un nuevo objeto `Position` cuyos valores son la suma de los ejes X e Y de los objetos `Position` pasados como parámetro. Si uno de los 2 parámetros es `null`, entonces debe lanzar una `NullPointerException` sin mensaje.
- **`distance`**: Calcula la distancia euclídea entre la posición que invoca el método y la posición recibida como parámetros. Si el parámetro es `null`, entonces devuelve 0.

Sprite (enumeración)

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

Wall (clase)

El valor por defecto para el atributo `pathable` es `false` y para sprite es `Sprite.WALL`.

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). Ya te la damos hecha. Básicamente tiene un objeto de tipo `Level` e invoca sus métodos.

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, puesto que están realizadas con el framework libgdx que no es objeto de estudio de esta asignatura.

Para ejecutar esta vista, verás que la ventana de Gradle de IntelliJ muestra dentro de `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.

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.

Evaluación

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

Requisito imprescindible para evaluar la práctica

Tipo de test	Comentarios
90 `"sanity"`	Estos test aseguran que la parte crítica del esqueleto del programa es respetada. Para probarlos haz: `Gradle → verification → testSanity` **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"`, `"advanced"` y `"expert"`).

Tipos de test y peso en la evaluación

Tipo de test	Peso	Comentarios
173 `basic`	70%	Estos test comprueban que los métodos básicos son funcionalmente correctos. Para probarlos haz: `Gradle → verification → testBasic` La nota se calculará a partir de la siguiente fórmula: (#test_basic_pasados / #test_basic) * 70%
25 `advanced`	17%	Estos test comprueban que los métodos avanzados son funcionalmente correctos. Muchos de estos test utilizan Mockito para emular ciertos comportamientos. Para probarlos haz: `Gradle → verification → testAdvanced` La nota se calculará a partir de la siguiente fórmula: (#test_advanced_pasados / #test_advanced) * 17%
13 `expert`	13%	Estos test comprueban que los métodos expertos son funcionalmente correctos. Estos test no simulan nada y, por lo tanto, requieren que la Práctica esté muy avanzada. Para probarlos haz: `Gradle → verification → testExpert` La nota se calculará a partir de la siguiente fórmula: (#test_advanced_pasados / #test_expert) * 13%

`Gradle → verification → test` ejecuta todos los test. **Si se superan los 301 test, la nota de la Práctica es un 10**.

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 `PACman` 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 Pràctica es el **29 de enero de 2023** antes de las 23:59. Cualquier Pràctica entregada más tarde será considerada como no presentada.