Proyecto 1 UT7 -- InvasoresFx

Objetivos

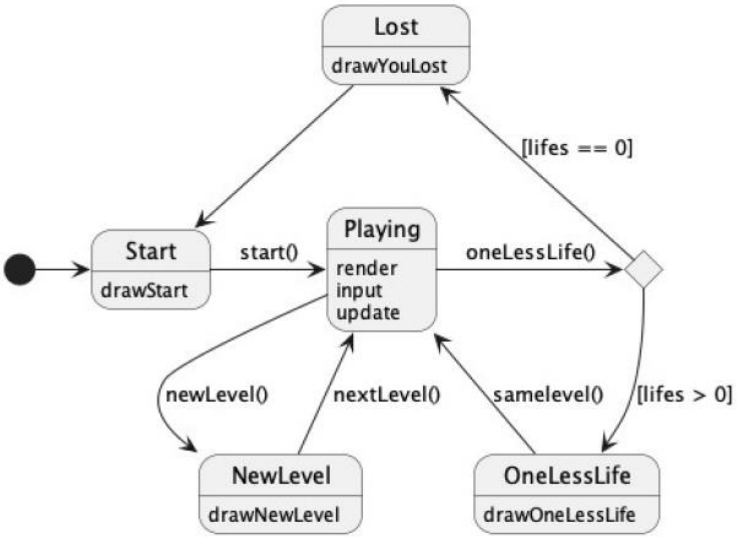
El objetivo de este trabajo es desarrollar programas aplicando características avanzadas de los lenguajes orientados a objetos. Concretamente utilizar los conceptos de herencia, clases abstractas e interfaces y explicarlos adecuadamente.

Antes de empezar

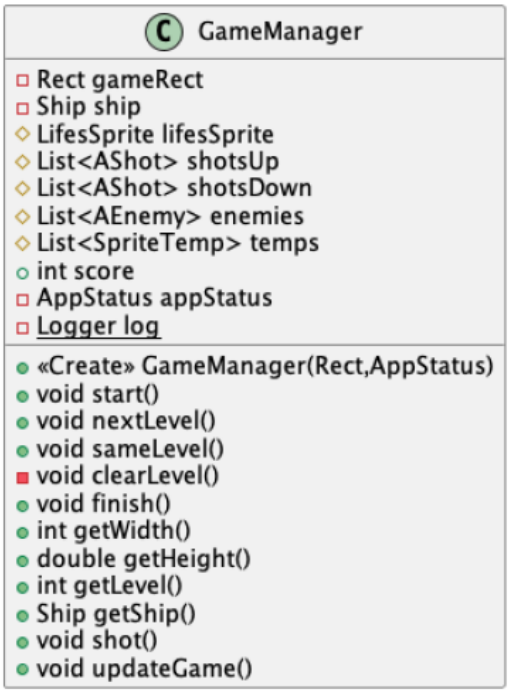
- Este ejercicio se realizará por parejas o individual.
- El proyecto de partida está en el Moodle.
- **Cada día** que trabajéis en el proyecto debéis hacer **al menos un commit** con un comentario.
- Se anulará automáticamente la corrección del ejercicio y se **evaluará con un 0 si** se detecta que **ha sido copiado** o dejado copiar a algún compañero/a
- El profesorado podrá convocar a cualquier alumno/a para **defender oralmente** el proyecto.

Introducción

Se va a facilitar el código inicial de un juego llamado **InvasoresFX** basado en el clásico **SpaceInvaders** sobre el que hay que trabajar. Un juego puede verse como una animación en la cual se puede interactuar y puede reaccionar a los eventos que se produzcan. No es el objetivo de esta práctica profundizar en su funcionamiento, pero se va a contextualizar el proyecto.

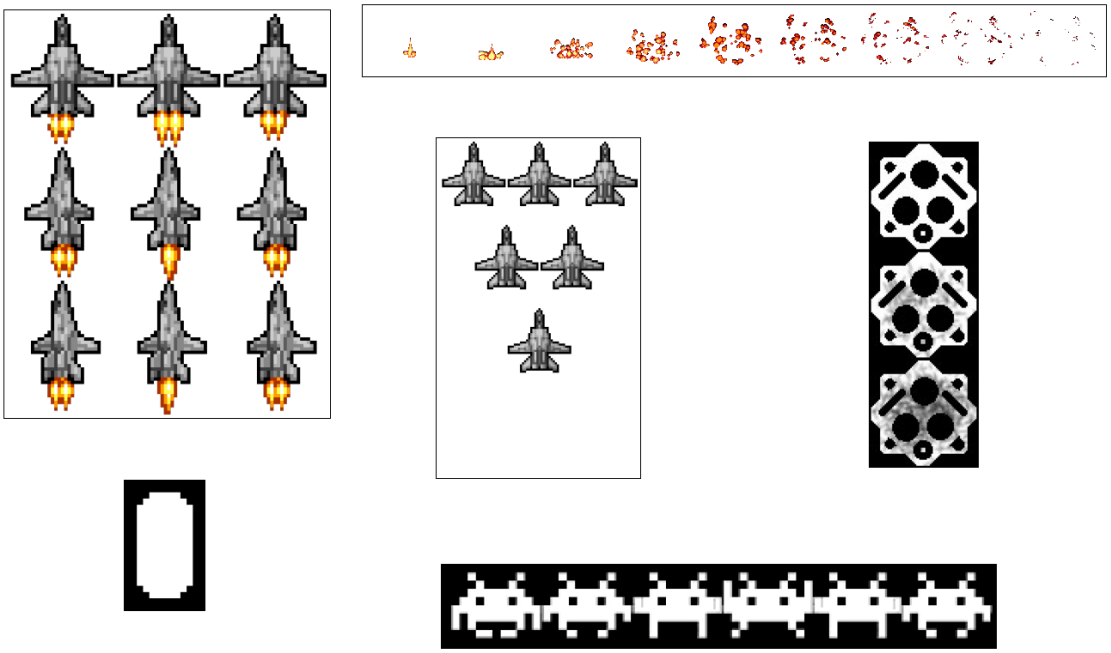


Durante la ejecución del juego, se ejecuta de forma continua el conocido como *game-loop* en el cual se renderiza el juego, se procesan las entradas y se actualiza el juego en consonancia.

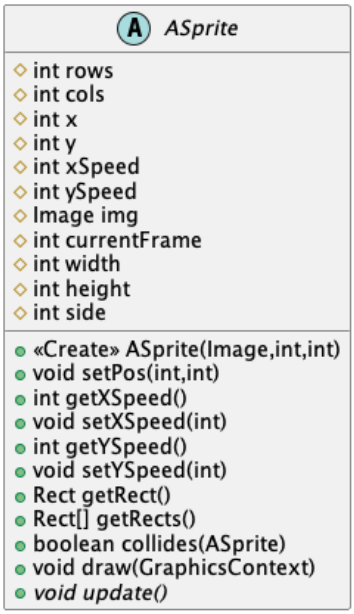

GameManager
Rect gameRect
Ship ship
LifesSprite lifesSprite
List<AShot> shotsUp
List<AShot> shotsDown
List<AEnemy> enemies
List<SpriteTemp> temps
int score
AppStatus appStatus
Logger log
«Create» GameManager(Rect,AppStatus)
void start()
void nextLevel()
void sameLevel()
void clearLevel()
void finish()
int getWidth()
double getHeight()
int getLevel()
Ship getShip()
void shot()
void updateGame()

```java
public void updateGame(){

    //Detección de colisión entre balas
     for (Iterator<AShot> itShotUp = shotsUp.iterator(); itShotUp.hasNext(); ) {...}

    //Detección de colisiones de los enemigos con los disparos del protagonista
    for (Iterator<AShot> itBullet = shotsUp.iterator(); itBullet.hasNext(); ) {...}

    //Actualización del protagonista
    if (ship.isAlive()) {...}
    //Actualización de los sprites temporales
    for (int i=temps.size()-1;i>=0;i--) {...}
    //Actualización de los enemigos
    for (ASprite enemy : enemies) {...}
    //Generación de nuevos enemigos
    List<AEnemy> newList = new ArrayList<>();
    for (ASprite enemy : enemies) {...}
    enemies.addAll(newList);
    //Comprobación del estado de la partida
    if (!ship.isAlive()&&(temps.size()==0)){...}
    //Comprobación del estado de la partida
    if ((enemies.size()==0)&&(temps.size()==0)){...}
    //Detección si los disparos de los enemigos han impactado con el protagonista
    //o se han salido del área se juego
    for (Iterator<AShot> itBullet = shotsDown.iterator(); itBullet.hasNext(); ) {...}
    //Detección si los enemigos han impactado con el protagonista
    //o se han salido del área se juego
    for (Iterator<AEnemy> itSprite = enemies.iterator(); itSprite.hasNext(); ) {...}

    //Generación de disparos por parte de los enemigos
    int n = shotsUp.size() - shotsDown.size();
    ArrayList<AEnemy> shooterEnemies = new ArrayList<>();
    for (AEnemy enemy : enemies) {...}
    if (shooterEnemies.size()>0){...}
}
```

La clase *GameManager* contiene los atributos más importantes para el funcionamiento del juego, así como métodos relevantes como *updateGame* cuya comprensión es importante.

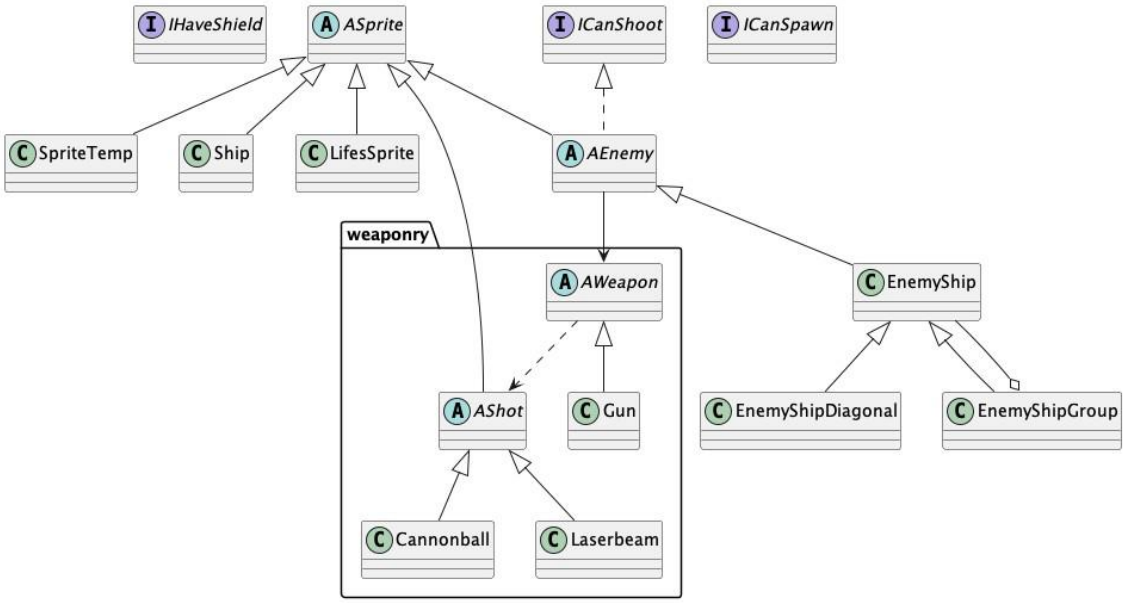


Un *sprite*, como imagen de dos dimensiones, contiene la representación de un objeto en diferentes estados que se suele utilizar para realizar animaciones, pero no tiene por qué. En el *sprite* del protagonista tenemos la primera fila cuando va al frente, la segunda a la derecha y la tercera a la izquierda. Todas las representaciones de la explosión están en una fila, mientras que las vidas están en una columna. La animación de los enemigos y el *sprite* del láser se ha puesto sobre un fondo negro para su visualización en este documento.

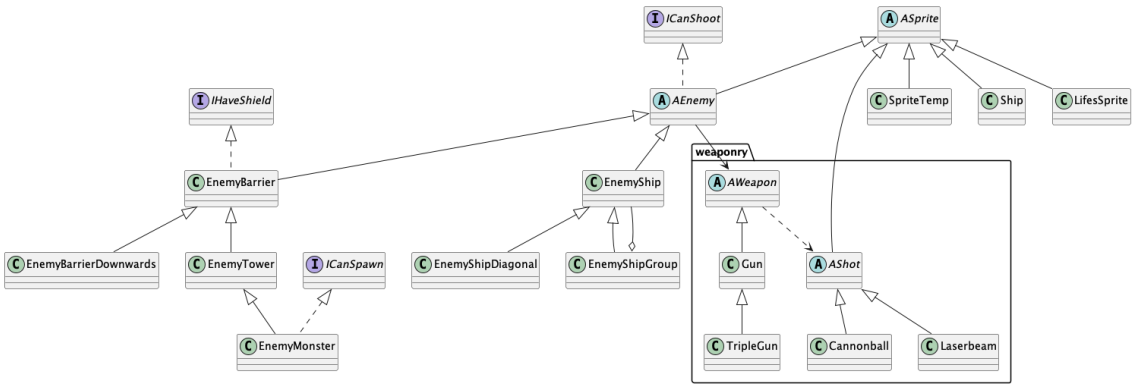


La clase *ASprite* facilitada contendrá entre otros atributos el número de filas y columnas que hay en la imagen. Es necesario comprender los atributos y métodos que contiene dado que el resto de clases de tipo *sprite* derivarán de esta.
Los *sprites* que se tienen que desarrollar pueden implementar interfaces existentes o creadas. La creación de nuevas interfaces puede influir en el resto de la aplicación como el *updateGame* del *GameManager* que habrá que analizar y pedir permiso para cambiar.

A continuación, se puede ver el esquema de clases facilitado y un ejemplo de esquema desarrollado. No se tienen por qué copiar las funcionalidades que se muestran, se pueden desarrollar otras nuevas. Por ejemplo, se puede crear una nave enemiga con escudo o que genere otras naves o tenga un arma distinta o con balas resistentes.



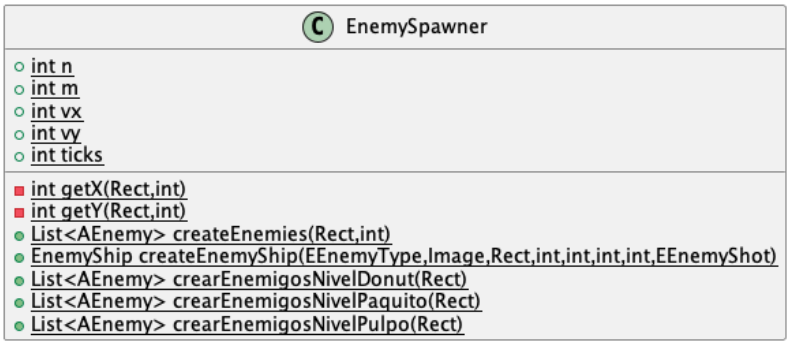
-Esquema de las clases facilitadas-



-Posible esquema que se puede llegar a desarrollar

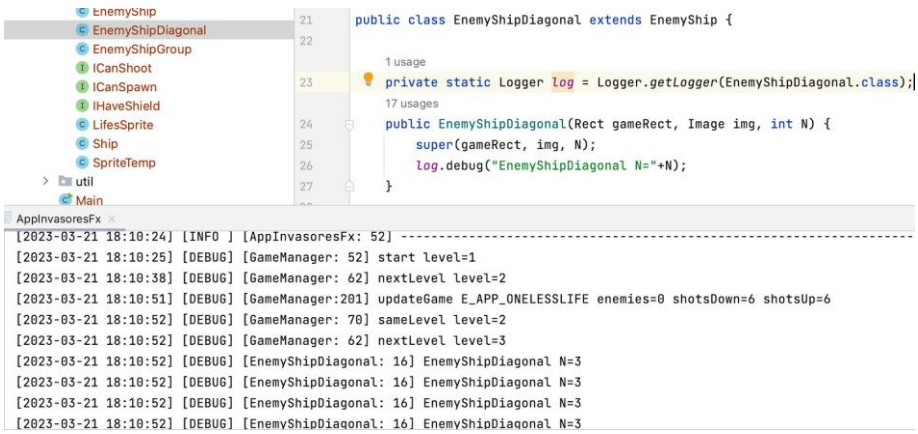
En los métodos *start* y *nextLevel* de *GameManager* se utiliza la clase *EnemySpawner* para generar los enemigos del primer nivel que se ejecuta o de los siguientes. El método principal *createEnemies* tiene como parámetros la referencia al rectángulo de juego y el nivel que se va a jugar.

Considerad también las constantes definidas en *AppConsts* como LEVELS que indica el número de niveles distintos implementados e INIT_LEVEL que sirva para empezar en el nivel que se desee en vez de empezar en el nivel uno (muy útil cuando estéis desarrollando un nivel). Volviendo a las explicaciones de la clase *EnemySpawner* esta contiene atributos que bien se podrían haber definido como constantes. Los atributos n y m se utilizan para dividir el tablero en celdas, de esta forma con los métodos *getX* y *getY* se puede traducir una coordenada de 0 a n o m en una posición x o y. Intentad entender el método *createEnemyShip* y como se utiliza para crear distintos niveles. Como se pide en el trabajo a desarrollar, habrá que implementar dos nuevos niveles y para ello habrá que agregar código a la clase *EnemySpawner*.



Logs

Para la depuración de la aplicación esta se puede ejecutar paso a paso o hacer uso de Log4j. Para poder utilizar el sistema de *logging* en una clase se debe definir un atributo estático de la clase *Logger* pasándole como argumento el nombre de la propia clase seguido de un punto y el atributo *class* al método *getLogger*. Para registrar un log se puede utilizar el método *debug* de la instancia log con el mensaje que se considere. Agregad los *imports* correspondientes. Observad la siguiente imagen a modo de ejemplo.



Tarea

Modificar el código base del juego facilitado para cumplir con los objetivos marcados. Se debe implementar alguna de las interfaces existentes o una nueva validada antes por el profesor además de hacer uso de las clases abstractas y la herencia.

Programación:
Hay que crear **dos enemigos** con algún comportamiento distinto en el que al menos un enemigo implemente una interfaz o **un enemigo que implemente al menos una interfaz** y otro objeto con alguna característica especial. En caso de querer implementar una nueva funcionalidad se debe pensar en las implicaciones que tiene en el resto de clases y si son asumibles. Los nuevos enemigos deberán heredar de *AEnemy()* y además tener alguna característica obtenida de los Interfaces (*ICanSapwn, IHaveShield, ICanShoot*...). Se podrá crear una nueva interface, según imaginación del alumno.

También hay que crear **al menos dos niveles de juego** nuevo en el que aparezca alguna instancia de estas clases. Haced que al ejecutarse el juego empiece en los niveles diseñados. El código facilitado, los enemigos y niveles que hay deben seguir funcionando.

Si el trabajo es individual se creará un nivel y un tipo de enemigo nuevo.

Memoria:
Se deberá crear una memoria que explique los cambios realizados; es aconsejable el uso de pantallazos para facilitar la documentación de la misma. La extensión de la memoria será de entre 5 y 8 páginas.

Entrega

La entrega del proyecto será a través de **Github**.
El formato de entrega de la **memoria** será en **pdf** a través del **Moodle**. En la portada de dicha memoria deberán ir los nombres y apellidos de los integrantes de la pareja y el enlace del repositorio Github donde está realizado el proyecto.
Si el proyecto se hace individual la portada contendrá un único nombre y apellidos además del enlace al repositorio de Github.
Fecha límite entrega: 7 marzo 23:59.

Rúbrica

Suspenso	Si no se hace uso correcto de las clases abstractas, interfaces y herencia o no se sabe explicar de forma que demuestre su uso. Si no se explica correctamente en la memoria el código generado o modificado. Si no se define en la fase inicial qué se va a desarrollar y qué implicaciones se prevén.
Aprobado	Si hay algún problema en el desarrollo de algún método y no ejecutarse la aplicación de forma correcta a cómo debería, pero los conceptos de POO se aplican correctamente y se saben explicar de forma adecuada en la memoria. Si puede ejecutarse correctamente y se explica bien, pero las clases generadas son adaptaciones simples del código facilitado. Si el código es bueno pero el vídeo aun sirviendo para explicar los conceptos es de mala calidad o no cumple con los requisitos.
Notable	Si se hace uso de la POO de forma correcta con código novedoso y se explica adecuadamente en la memoria estando estructurada y presentada de manera correcta.
Sobresaliente	Si se demuestra el dominio de la POO en cuanto a la codificación y las explicaciones de la memoria, la calidad de la misma es excepcional (índice, explicación bien estructurada, lenguaje adecuado, aportación de imágenes correcta con subtítulos…). Se crea alguna solución brillante que funciona. Desarrolla más objetos y niveles de los requeridos que aportan. Se ha pensado en nuevas interfaces que se han utilizado de forma correcta y se pueden reutilizar. Se ha pensado en las implicaciones en el código restante del juego y se ha sabido modificar de correctamente.