

Proyecto **Asteroids**

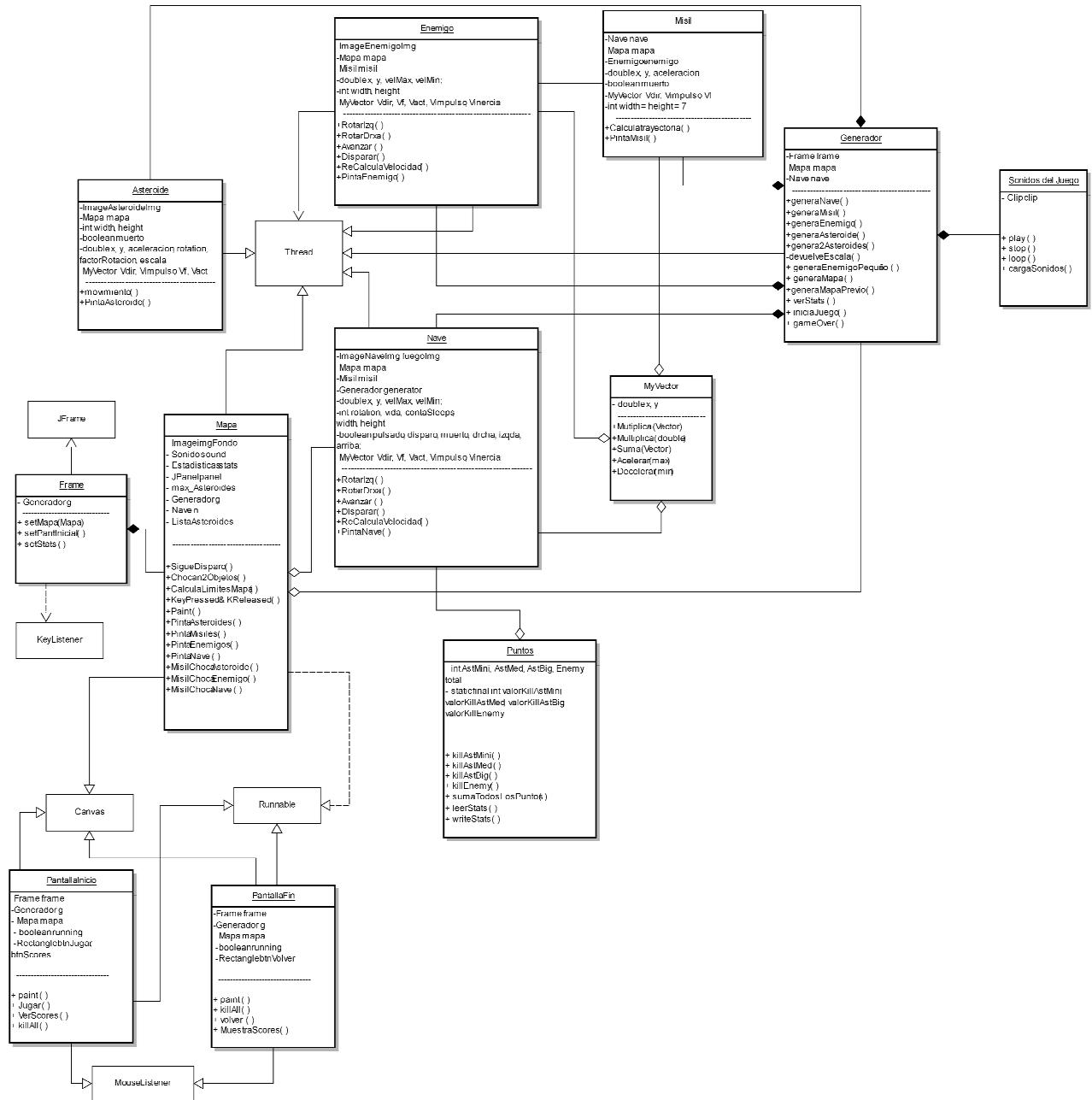
**UML**

Servicios y procesos

Álex Pérez Santiago  
DAM2A14 – 2016

## El UML

A continuación se muestra la imagen del diseño UML resultante del juego Asteroids.



## Razonamientos

En este apartado trataremos de explicar el razonamiento seguido para llevar a cabo la arquitectura de software del programa *Asteroids*.

La clase principal que alberga el `main` del programa es **Launcher**, en la cual se inicia un objeto **Generador** y este se inicializa mediante el método `start` propio al heredar de `Thread`. Esta clase **Generador** inicializa la pantalla inicial del juego, con un mapa en el que aparecen asteroides de fondo y un botón *Jugar* que dará comienzo al juego al ser pulsado. El contenedor del juego será la clase **Frame** que extiende de `JFrame` y a su vez hereda de `KeyListener` para controlar los eventos de teclado de la partida.

Una vez iniciado el juego, en el método `run` de este hilo se generará la pantalla inicial, se creará el Mapa y se iniciará el hilo correspondiente. También usaremos esta misma clase para generar los Asteroides, la Nave, el Enemigo, Misiles, los sonidos del juego y, finalmente, para mostrar la pantalla final una vez el juego haya concluido, gracias al método `gameOver`. Comentaremos el resto de métodos de esta misma clase más adelante en la explicación del resto de clases del juego.

Con el juego ya en marcha, la clase **Mapa** controlará todos los eventos y acciones que suceden en el mismo juego. Para ello hemos hecho que esta clase herede de `Canvas` para poder pintar en el mapa los asteroides, enemigos, nave y misiles. Al ser esta clase la más extensa del juego comentaremos brevemente sus principales métodos.

En el método `run` de este hilo creamos un `BufferStrategy` de 2D que será suficiente para pintar todos los objetos del juego y se crea un bucle que terminará cuando lo haga el propio juego, es decir, cuando perdamos todas las vidas de la Nave. En este bucle se irán pintando todos los objetos, así como la información requerida por pantalla, a través del método `paint` y del objeto `Graphics2D`.

Se ha implementado el método `chocaObjeto` al que aplicamos sobrecarga de objetos ya que lo usaremos con el mismo propósito tanto para *Nave* como para los *Enemigos* del mapa. Este método se encarga de comprobar las colisiones de dichos objetos con cualquier otro objeto del mapa. Otro método importante es `sigueDisparo`, este es invocado por cada hilo de *Misil* disparado en el mapa, tanto para misiles de la nave como para misiles disparados por los enemigos. Así se comprueba si dicho misil disparado choca con algún objeto o sale del mapa. En caso afirmativo el *Misil* (hilo) será destruido al igual que el objeto con el que choca.

Como hemos comentado anteriormente, en el bucle del `run` de esta clase se irán creando los distintos objetos que forman parte del juego. En el caso de los *Asteroides*, contamos con un atributo del propio mapa que corresponde con el número máximo de asteroides que puede albergar según el nivel actual. Dentro del bucle se hace la llamada al objeto `generador` encargado de instanciar e iniciar los distintos asteroides en función del máximo permitido. Así damos paso a la clase **Asteroid** a continuación.

Para esta clase, **Asteroid**, se han implementado 2 constructores. Uno que instanciará *Asteroides* grandes y la otra que instanciará asteroides más pequeños en función de las características del asteroide al que preceden. El movimiento de los objetos de esta clase se consigue mediante la aplicación de vectores.

La clase **MyVector** es la encargada de definir el tipo de movimiento que siguen todos los objetos del juego. Para ello, esta clase cuenta con una serie de métodos que nos permiten calcular la suma y multiplicación de vectores, extraer su módulo y reajustarlo para acelerar o decelerar la velocidad de desplazamiento.

La clase **Misil** también hace uso de los vectores para moverse a través del mapa, de un modo muy similar al de los asteroides, comentado anteriormente. En esta clase también contamos con 2 constructores: uno para el misil que dispara la Nave y otro para el misil disparado por el/los enemigos. En el mismo constructor se define ya el vector final y la trayectoria del disparo, ya que este no varía a lo largo de su trayectoria. Una vez se produce un disparo, se inicia el hilo de esta clase y se convoca al método `calculaTrayectoria` que definirá las consecutivas posiciones del misil a lo largo de su vida y al método de *Mapa* `sigueDisparo` ya comentado en su clase.

La clase **Enemigo** contiene similitudes con la de Nave. En un principio se pensó en hacer que *Enemigo* heredase de Nave (por ser una de las primeras clases implementadas en el desarrollo del proyecto), pero sus notables diferencias provocaron en la derivación de 2 clases distintas. También se planteó la posibilidad de hacer una clase común para ambas, más genérica, y hacer que estas 2 clases heredasen los métodos comunes pero también se descartó esta idea por las razones que se exponen a continuación:

1. Evitar sobresaturar de clases el proyecto
2. Las diferencias de comportamiento de ambas clases
3. Simplicidad general
4. Incrementar la cohesión y evitar dependencias innecesarias
5. Aumentar la coherencia de clases

A continuación daremos una explicación de la clase Nave al ser más extensa y complicada que la del Enemigo.

La clase **Nave** es una de las clases más relevantes del juego ya que es el objeto que representa al jugador en la partida. Esta clase hereda de *Thread* y depende de los eventos de teclado capturados por el *KeyListener* de la clase XXXXXX. Según el evento de teclado recogido, este provocará una llamada al método encargado de modificar la posición o rotación de la *Nave* para ello contamos con los métodos `subeRotation` y `bajaRotation` encargados de modificar el ángulo de rotación de la nave y el método `impulsaNave` que modificará el vector impulso para propulsar la nave según el ángulo de rotación que tenga en ese momento. El método `recalculaVelocidad` nos sirve para hacer los cálculos necesarios del vector final de la nave y definir su velocidad máxima y mínima imitando así el comportamiento del juego original. Con el método `disparar` se llama al método del objeto *Generador* que instanciará un nuevo hilo de *Misil*. Por último, utilizamos el método `cargaImg` en el constructor para cargar las imágenes correspondientes a la nave y el método `pintaNave`, que recibe el objeto *Graphics2D* del mapa, utilizado para dibujar la nave en el *Canvas* del mapa de igual forma que hacemos con los asteroides y enemigos.

La clase **Puntos** se ha creado principalmente para mantener toda la lógica de la suma de puntos a parte y para la lectura y escritura de puntuación del jugador de la partida. Para ello, contamos con los métodos propios que son llamados cuando se elimina un asteroide o un enemigo y las constantes con los puntos obtenidos por ello. También se incluye el método de lectura del ranking de puntuaciones `leerStats` que, en este caso, se ha optado por guardar los datos en un simple fichero CSV y el método de escritura `writeStats` que usaremos para añadir la puntuación de la última partida en ese mismo fichero.

La clase **Sonidos** consta de un constructor y un manejador `AudioInputStream` para los diferentes objetos `Clip` a través de los cuales usaremos los métodos `play`, `stop` y `loop` para reproducir los sonidos que añadirán vida y jugabilidad a la partida. La clase `Generador` será la encargada de hacer las llamadas a estos métodos según corresponda.

La clase **PantallaFinal (Scores)** nos mostrará los resultados de los jugadores ordenados por puntuación, que extraerá dichos datos de un fichero CSV donde se guardan los puntos al finalizar la partida. También podremos volver a la pantalla inicial y empezar un nuevo juego si lo deseamos.

La clase **PantallaInicial** nos sirve para mostrar un mapa inicial de cómo será el juego, incluye 2 zonas que usaremos como botones para: mostrar el ranking de jugadores, o bien, empezar la partida.

## Conclusiones

Con el desarrollo de este proyecto se ha podido comprobar la importancia de una buena arquitectura del software para proyectos de cierta envergadura y se han intentando aplicar y aprovechar al máximo las características de la orientación a objetos para facilitar el desarrollo del juego *Asteroids*, así como lo aprendido durante el curso sobre los *Threads* de Java.

No obstante, me hubiera gustado poder contar con un poco más de guía a la hora de desarrollar el proyecto, ya que de ese modo considero que el tiempo de desarrollo podría haber sido también menor. Sin embargo, la falta de guía nos ha obligado a esforzarnos mucho más, buscar información y, en consecuencia, forzar también el autoaprendizaje que todo programador necesita.

Si volviera a empezar el proyecto de nuevo, sin duda dedicaría más tiempo al diseño de la arquitectura, apoyándome en todo lo aprendido con esta práctica. Ya que como he podido comprobar he tenido que rehacer varias veces la estructura por la falta de un diseño claro y estructurado desde el inicio.