Genetic Kingdom

by Nicolás Florez, Tamara Cajiao & Justin Solano.

Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computadores

Algoritmos y Estructuras de Datos II (CE 2103)
I Semestre 2025

Tabla de contenidos:

Introducción:

Este documento detalla el desarrollo del juego Genetic Kingdom, un Tower Defense creado en C++ para el curso Algoritmos y Estructuras de Datos II (CE2103) del Instituto Tecnológico de Costa Rica. A lo largo de este documento se explicarán los elementos clave del juego, incluyendo su mecánica de defensa de un castillo medieval contra enemigos que evolucionan mediante algoritmos genéticos, el sistema de torres y mejoras, y la implementación del Path Finding (A*) para el movimiento estratégico de los enemigos.

Descripción General:

Genetic Kingdom es un juego de estilo Tower Defense desarrollado en C++ como parte del curso Algoritmos y Estructuras de Datos II (CE2103) del Instituto Tecnológico de Costa Rica. El juego está ambientado en la Edad Media y tiene como objetivo principal defender un castillo de oleadas de enemigos que evolucionan utilizando algoritmos genéticos. Para ello se plantea el uso de torres con un sistema de mejora por cobro a base de una moneda definida por el grupo. Además los enemigos tienen la capacidad de escoger un camino con el uso de Path Finding (A*), esto debe ser implementado de modo que no se interrumpan todos los caminos de los enemigos al castillo.

Implementación de la solución:

ID	DESCRIPCIÓN	SOLUCIÓN	SOLUCIONES ALTERNATIVAS
001	El mapa es una cuadrícula de tamaño fijo definida por el grupo de trabajo. El usuario puede colocar torres en cualquiera de los cuadros del mapa asegurando que no bloqueen todos los posibles caminos hacia el puente. El mapa tiene un único punto de ingreso de los enemigos y está en el lado contrario al puente del castillo	El mapa es una cuadrícula implementada en las clases Map y Tile. El punto de entrada está fijo en el extremo opuesto al castillo. Las torres se colocan en celdas libres, y el sistema de Pathfinding A* asegura que siempre exista al menos un camino disponible para los enemigos.	En lugar de una cuadrícula estática, se podría implementar un sistema de "plantillas de mapa". El juego incluiría varias plantillas de cuadrícula predefinidas (pero fijas en tamaño) que se cargan al inicio de la partida. Esto añade variedad visual y de diseño de nivel sin abandonar la restricción de la cuadrícula fija. Las plantillas definirían la posición del castillo, el punto de entrada de los enemigos y los obstáculos predeterminados. **Problema Encontrado:** La implementación de 'plantillas de mapa' generó problemas inesperados con el Pathfinding A*. Si bien las plantillas eran fijas en tamaño, la colocación dinámica de obstáculos (torres) por parte del jugador creaba escenarios donde el A* requería una adaptación constante a las plantillas. Esto resultaba en un alto costo computacional y causaba ralentizaciones en el juego, especialmente con múltiples enemigos y torres. Además, surgieron errores en la validación de caminos bloqueados, donde el juego permitía la colocación de torres que aislaban porciones del mapa en algunas plantillas. Por eso se escogió la solución de cuadrícula estática.

002	Cada torre tiene los siguientes atributos: daño, velocidad, alcance, tiempo de regeneración del poder especial, tiempo de recarga de ataque. Las torres atacan a los enemigos cuando están a su alcance. Con cada muerte de enemigo, se considera su categoría y tipo para devolver cierta cantidad de oro al jugador. La cantidad de oro retornado debe ser calculado de forma justa y consistente. El grupo de trabajo define esto.	Las clases ArcherTower, MageTower y ArtilleryTower definen las torres. Cada una tiene atributos diferenciados. Las torres atacan enemigos dentro de su alcance. El sistema económico está en construcción; el diseño contempla oro otorgado según el tipo de enemigo eliminado.	La instrucción es clara y no deja mucho espacio para alternativas.
003	Todas las torres tienen 3 upgrades. Cada upgrade aumenta el daño que pueden causar y cada torre tiene ataques especiales que ocurren cada cierto tiempo con una probabilidad definida por los estudiantes.Los upgrades tienen un costo en oro. Cada upgrade es más caro que la anterior. Cada upgrade modifica los atributos en un valor definido por el grupo de trabajo.	Se planificaron 3 niveles de mejora por torre, incrementando daño y atributos. Cada mejora tiene un costo creciente en oro. El código estructura ya permite upgrades; la implementación de los ataques especiales con probabilidad está en desarrollo, definido por el grupo como 20-30% chance.	Implementar "ramas de mejora". En lugar de una simple progresión lineal de 3 niveles, cada torre podría tener 2 o 3 opciones de mejora por nivel. Por ejemplo, la torre de arqueros podría mejorar su velocidad de ataque o su alcance en el nivel 1. Esto permite al jugador especializar las torres según su estrategia. **Problema Encontrado:** La implementación de 'ramas de mejora' introdujo una complejidad significativa en la lógica del sistema de partidas. Cada torre requería almacenar qué rama específica de mejora había tomado el jugador en cada nivel, lo que aumentaba la cantidad de datos. Además, la interacción de las diferentes ramas de mejora entre sí generó bugs inesperados, donde ciertas combinaciones causaban comportamientos no deseados en las torres. El tiempo necesario para depurar estos

			problemas se consideró excesivo.
004	Hay 3 tipos de torres: ● Arqueros: bajo costo, alto alcance, poco daño, tiempo de recarga de ataque bajo ● Magos: costo medio, alcance medio, daño medio, tiempo de recarga de ataque medio ● Artilleros: costo alto, alcance bajo, daño alto, tiempo de recarga de ataque alto	Las tres torres están implementadas en archivos separados. Su comportamiento respeta las especificaciones: Arqueros (bajo daño, alto alcance), Magos (balanceados) y Artilleros (alto daño, bajo alcance). Cada clase gestiona sus atributos únicos.	La instrucción es clara y no deja mucho espacio para alternativas.
005	El jugador puede ir colocando las torres en cada lugar disponible. Al seleccionar un lugar disponible, puede escoger el tipo de torre que desea crear. Cada torre tiene un costo en oro.	El jugador selecciona celdas libres en la cuadrícula para colocar torres. Al seleccionar una celda, puede escoger entre las tres torres disponibles. Cada torre tiene un costo en oro, el cual será verificado antes de su colocación mediante el sistema económico del EconomySystem.	Implementar un sistema de "previsualización de colocación". Antes de gastar oro, el jugador puede seleccionar una torre y ver su alcance y área de efecto (si la tiene) superpuestos en la cuadrícula. Esto ayuda a tomar decisiones de colocación más informadas. **Problema Encontrado:** El sistema de 'previsualización de colocación', aunque útil, afectó negativamente al rendimiento del juego. Calcular y renderizar el alcance y el área de efecto de las torres en tiempo real, especialmente con múltiples torres en el mapa, consumía muchos recursos de la CPU. Esto provocaba caídas en la velocidad de fotogramas, especialmente en las oleadas con gran cantidad de enemigos. Optimizar este sistema para un rendimiento aceptable se

			consideró fuera del alcance del proyecto.
006	Los enemigos aparecen por oleadas. Cada oleada es una generación que evoluciona. Los enemigos pueden ser los siguientes: ● Ogros: son el enemigo más básico. Son resistentes a los arqueros y débiles contra la magia y la artillería. Son lentos. ● Elfos Oscuros: son resistentes a la magia, pero débiles a los arqueros y a la artillería. Son muy rápidos ● Harpías: solo pueden ser atacadas por magia y arqueros. Tienen una velocidad intermedia ● Mercenarios: son débiles a la magia, pero resistentes a arqueros y artillería. Cada enemigo tiene atributos como: ● Vida ● Velocidad ● Resistencia a flechas ● Resistencia a la magia ● Resistencia a la artillería	Los enemigos se gestionan en WaveManager. Cada tipo está representado por clases específicas (Ogre, Harpy, etc.) y poseen atributos: vida, velocidad y resistencias a cada torre. Las debilidades y fortalezas están codificadas según el tipo, siguiendo las especificaciones.	Añadir un comportamiento de "formación" a los enemigos. En lugar de simplemente seguir el camino A*, los enemigos podrían agruparse en formaciones (línea, columna, etc.) que les proporcionen bonificaciones (ej., los ogros en la primera fila protegen a los elfos oscuros detrás). La formación podría cambiar dinámicamente según la disposición de las torres. **Problema Encontrado:** La implementación de 'formaciones' enemigas introdujo problemas complejos de IA. Lograr que los enemigos mantuvieran las formaciones de manera coherente mientras navegaban por el camino A* y evitaban las torres resultó ser un desafío. Los enemigos a menudo se dispersaban o se atascaban. Además, las bonificaciones de formación requerían un sistema adicional para rastrear la posición relativa de los enemigos, lo que aumentaba la complejidad del código.
007	Cada generación selecciona los individuos con el mejor fitness, los cruza e ingresa los nuevos individuos a la población. Pueden ocurrir	Se definió un Algoritmo Genético (GeneticAlgorithm).	Además de lo básico se podría implementar mejores aplicaciones para que el algoritmo genético luzca más a través de la partida, como

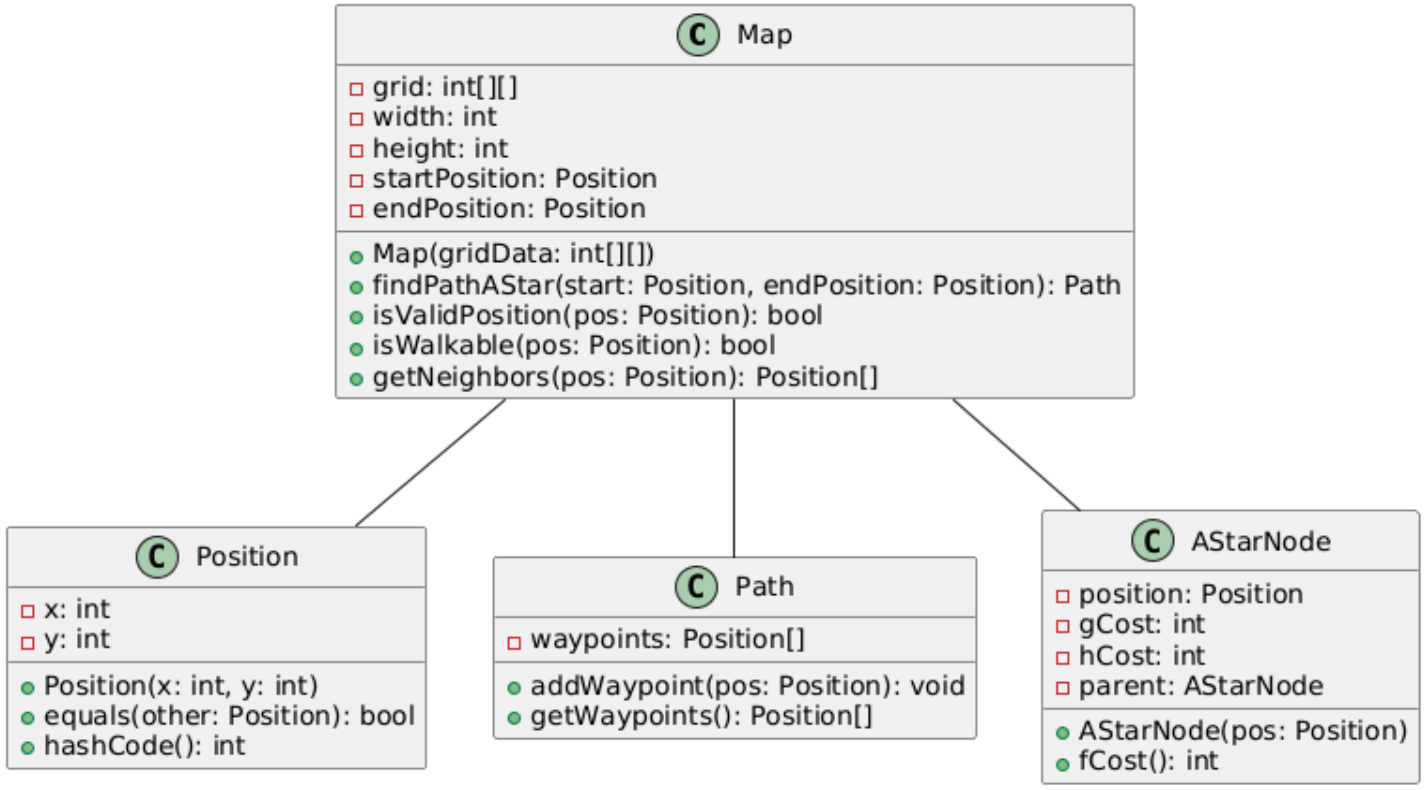
	mutaciones con cierto grado de probabilidad. El estudiante debe definir cada elemento del algoritmo genético y explicarlo adecuadamente en la documentación. Las oleadas son de tamaño variable y se generan con un intervalo parametrizable	Este componente se encarga de la valoración y evolución de los enemigos entre oleadas. Sus funciones principales incluyen: Evaluación de la población: Calcula promedios de atributos de toda la población para determinar la aptitud de cada enemigo. Selección de padres: Elige enemigos con al menos dos atributos por encima del promedio para reproducirse, es decir los padres de la siguiente generación. Cruzamiento (crossover): Combina atributos de dos padres para crear un nuevo enemigo, esto sumando ambos para sacar el promedio de estos. Mutación: Aplica cambios aleatorios a ciertos atributos del enemigo con una probabilidad definida del 50% (afectando 1 o 2 atributos en 1.5x).	dando posibilidades de enemigos de élite o similares. **Problema Encontrado:** El manejo de la lógica para esto sumado a la volatilidad que producía en las generaciones cruzadas al darles estadísticas demasiado altas provocaba una subida de dificultad innecesaria a fines del proyecto.

008	Los enemigos utilizan Pathfinding A* para encontrar el camino hacia el puente del castillo.	La estructura del Pathfinding A* está implementada en Pathfinding.cpp. Este sistema permite que los enemigos busquen el camino óptimo hacia el castillo, considerando obstáculos como torres. Asegura siempre dejar caminos abiertos conforme a la regla del requisito 001.	Introducir "puntos de interés" en el mapa que influyan en el pathfinding. Además del objetivo principal (el castillo), podría haber puntos en el mapa que atraigan a ciertos tipos de enemigos (ej., un pozo de salud que atrae a los ogros). Esto añade complejidad táctica al movimiento enemigo. Se consideró la posibilidad de que las torres pudieran ser obstáculos para los enemigos, sin embargo, esta idea no se terminó realizando pues, el comportamiento de los enemigos terminaría siendo muy lento visualmente, pues estos deberían estarse actualizando en todo momento en caso de que una torre fuera colocada en frente de estos. **Problema Encontrado:** La implementación de 'puntos de interés' en el mapa, si bien añadía complejidad táctica, generó problemas de rendimiento. El A* debía recalcular los caminos de los enemigos con frecuencia para tener en cuenta la atracción de los puntos de interés, especialmente si estos eran dinámicos o si varios puntos competían por la atención de los enemigos. Esto resultó en un mayor uso de la CPU y en ralentizaciones del juego. Además, la IA enemiga se volvía errática en algunos casos, con los enemigos oscilando entre el castillo y los puntos de interés de manera poco natural.

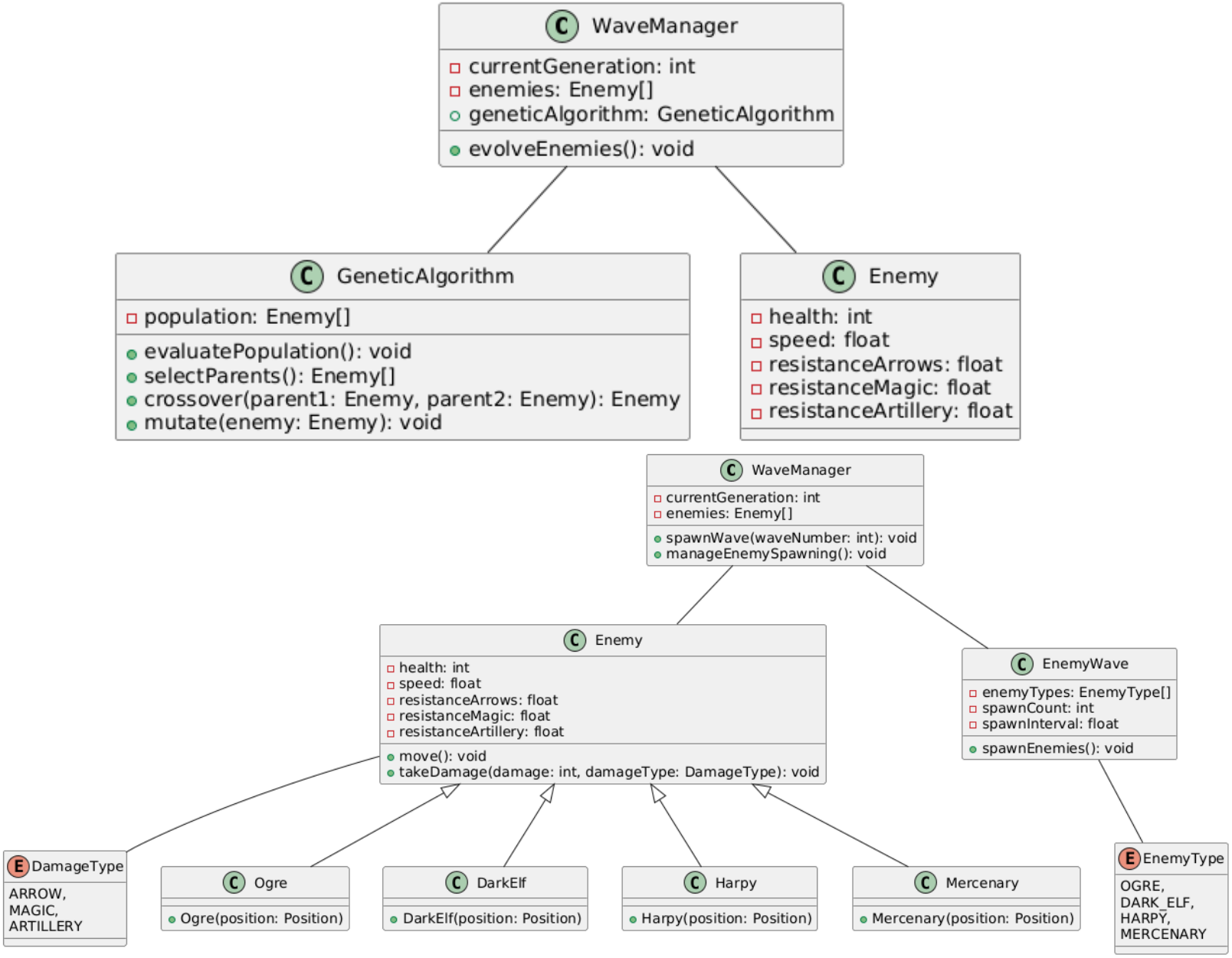
009	El juego muestra un panel con estadísticas como: ● Generaciones transcurridas ● Enemigos muertos en cada oleada ● Fitness de cada individuo de la oleada ● Nivel de cada torre ● Probabilidad de mutaciones y cantidad de mutaciones ocurridas	El juego incluye paneles (BottomPanel, SidePanel) que muestran información. La conexión visual en UI para mostrar fitness, muertes y mutaciones está en progreso, y se integrará al finalizar la lógica de oleadas.	Existía la posibilidad de generar un solo panel con dicha información o hacerlo de forma desplegable en base a lo seleccionado del mismo. **Problema Encontrado:** Se descartó la posibilidad de un solo panel con todas las estadísticas pues, en un solo panel la información estaría muy comprimida y saturada en un solo espacio. Además se descartó la posibilidad de paneles desplegables, justamente por la razón opuesta, no son tantas estadísticas las que se necesitan mostrar, por lo que es suficiente con un panel inferior y otro lateral.

Diseño General (Diagrama de clases con patrones de diseño implementados y POO):

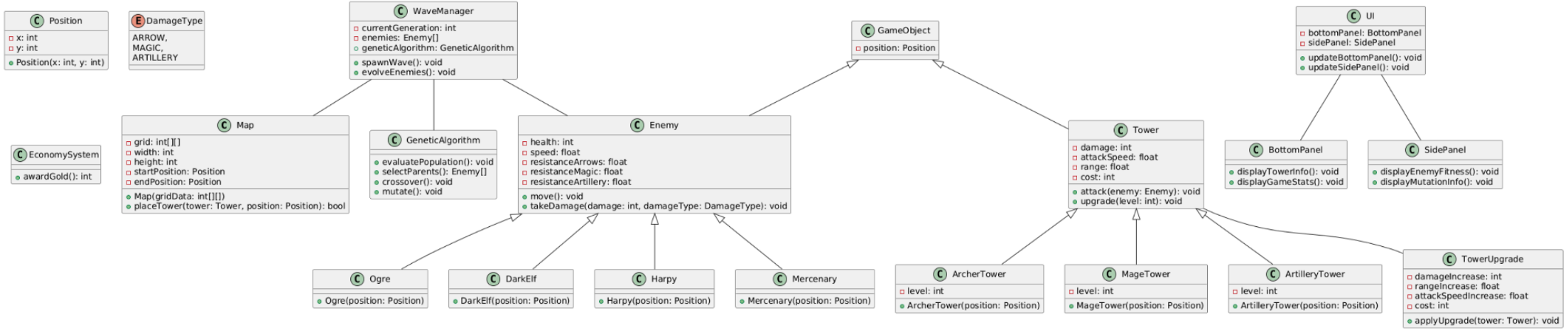
Mapa y A*



Wave Manager y Enemy



Vista General



Link repositorio:
NicoFJ09/Genetic_Kingdom_CE