Trabajo Práctico N°2
Informe de trabajo

ASIGNATURA
Algoritmos y Programación 3

TÍTULO
AlgoCraft

CORRECTOR
Joaquín

ALUMNOS
Joaquín Ordoñez
Juan Patricio Amboage

CICLO
2do cuatrimestre 2022

Mapa

El mapa consta de un tablero bidimensional de 10x10, siendo estas medidas preestablecidas por el juego de forma invariable para todas las partidas. La existencia del mapa queda regulada por su clase homónima, que funciona a modo de Singleton. Este patrón se presentó como ideal por dos motivos: el primero es que la clase Mapa como objeto es una única entidad de su tipo; y el segundo, que su gran demanda por varias de las otras clases se ve simplificada al acceder al mismo mediante un getter de clase, que ahorra el pasaje del mapa por parámetro a todas aquellas clases que lo necesiten.

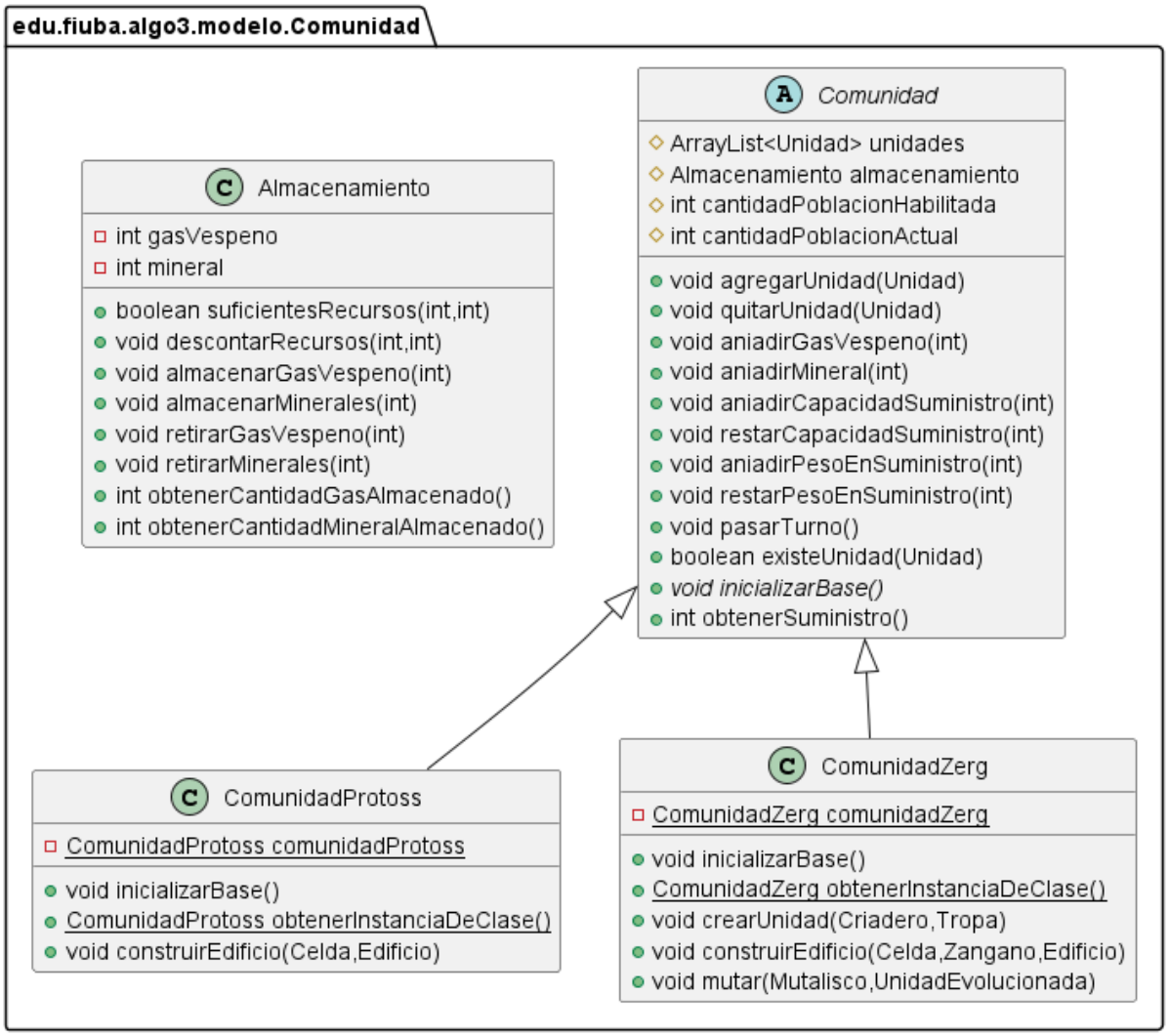
La clase Mapa que contiene al mapa, valga la redundancia, guarda a este último como atributo del tipo Arraylist de Arraylists. Es decir, hay un ArrayList con diez elementos (que representan las filas), que a su vez cada uno de los mencionados es un ArrayList con otros diez elementos (representando a las columnas). Los Arraylists internos son quienes finalmente contienen las celdas, representadas como objetos de la clase abstracta Celda.

Creación y formación del mapa: Se va realizando por partes. El primer paso será llamar al método denominado instanciarMapa(), que mediante dos ciclos for instancia todos los objetos de celda dentro del ArrayList doble. Por otro lado, en este mismo método, se crea una instancia de la clase GeneradorElementos, una clase compuesta a la que Mapa le delega las tareas de generar los edificios base de los jugadores así como la distribución de recursos al azar en el tablero.

Comunidades

Cuando hablamos de comunidad nos referimos a la globalización de todos los miembros de una raza, es decir, de los participantes del juego. Dado que en este caso se enfrentan dos razas, existe una comunidad para cada una, a las que representamos mediante ComunidadZerg y ComunidadProtoss. Como se mencionó anteriormente, son clases destinadas a controlar y contener a todas las existencias de las razas Zerg y Protoss, respectivamente. Ambas clases heredan de una

abstracta Comunidad, que reúne varios métodos y atributos en común, algunos de los cuales son redefinidos a conveniencia de las hijas. Cabe mencionar que al igual que con el mapa, las comunidades particulares (hijas) funcionan como singletons.

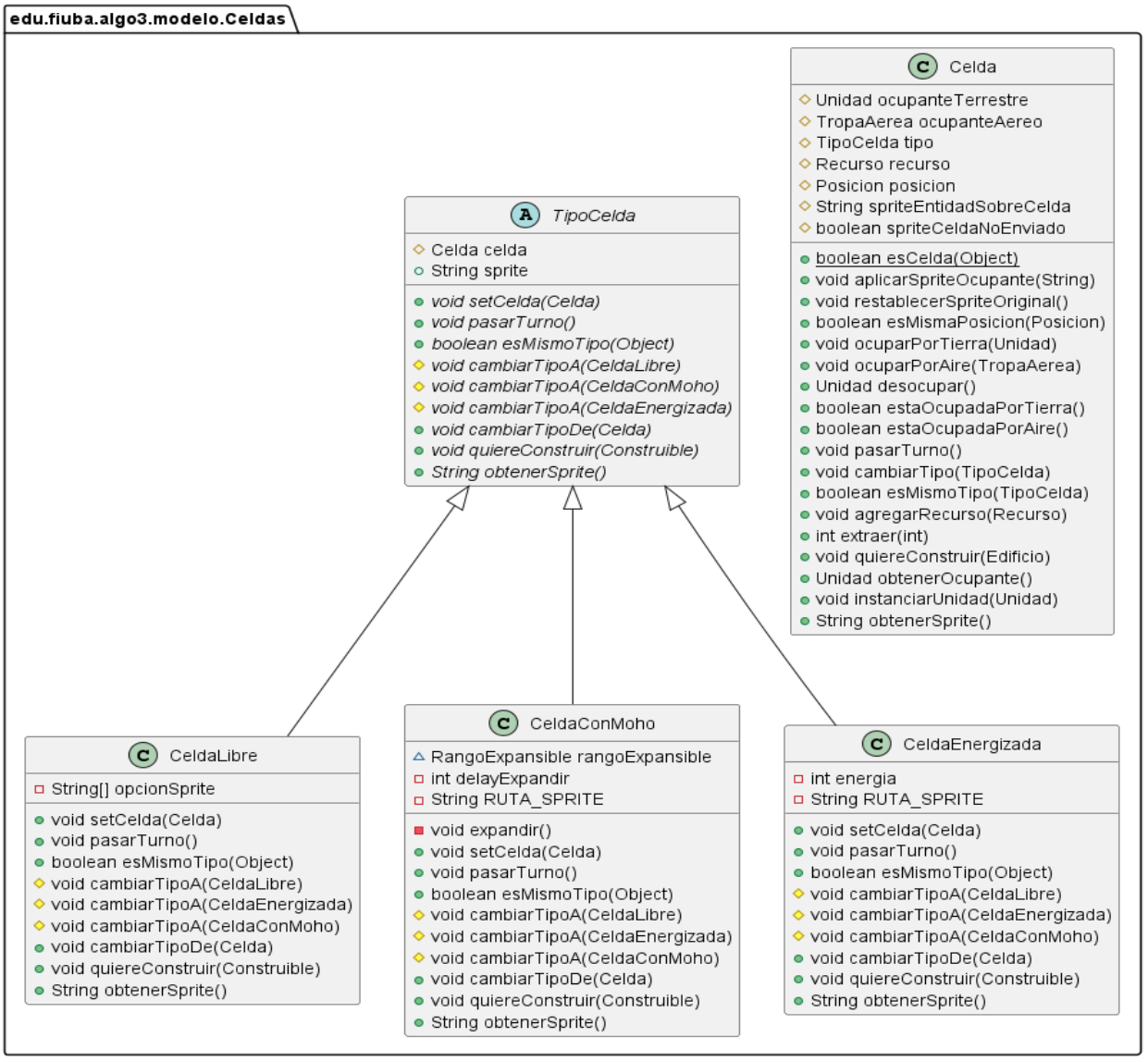


Como puede observarse en el diagrama, una comunidad agrupa a todas sus unidades dentro de un ArrayList, que representaría a la población total. También guarda una clase compuesta llamada Almacenamiento, a la que se le delegan las tareas de control de recursos de la raza (minerales y gas vespeno).

Celdas

Según la configuración preestablecida en nuestro juego, hay cien celdas en el mapa. Pueden ser del tipo: CeldaLibre, donde no reposa ningún tipo de fenómeno particular; CeldaConMoho, infectadas con moho, un fenómeno característico de la raza Zerg que le permite edificar sobre la celda; y por último, CeldaEnergizada, que se ve alcanzada por energía proveniente de un edificio energizador Protoss y concede la posibilidad de construir edificios para dicha raza.

La celda como tal siempre será un objeto Celda. Su tipo específico (las tres variantes descritas en el anterior párrafo) son manejadas mediante un patrón State dentro de Celda, como se muestra en el siguiente diagrama.



Continuando con la clase Celda, posee una serie de atributos que explican los contenidos que pueden haber en ella. Desde tropas que la ocupan por tierra o aire, hasta recursos.

Ocupante terrestre: Representa a una única unidad que puede posicionarse en dicha celda vía tierra. Puede ser una tropa o un edificio. Si hay una tropa, no puede haber un edificio, ni otra tropa. Y viceversa.

Ocupante aéreo: Al igual que con el ocupante terrestre, puede haber uno y solo uno. En este caso se hace referencia a aquellas tropas que tienen la capacidad de desplazarse por aire. Claramente, una tropa terrestre o un edificio no podrán posicionarse como ocupantes aéreos. Tampoco la tropa aérea podrá “bajar” a tierra, siempre se mantendrá volando desde que es creada hasta que deja de existir.

Tanto ocupante terrestre como aéreo pueden coexistir en la misma celda, lo que no pueden haber son dos ocupantes terrestres o dos ocupantes aéreos.

Recurso: Si la celda se eligiera como generadora de un recurso, se carga a este atributo. En tal caso las tropas pueden desplazarse sin inconvenientes (siempre que no haya otro ocupante). No así en el caso de los edificios, ya que solo los edificios recolectores podrán ser construidos si existiera un recurso.

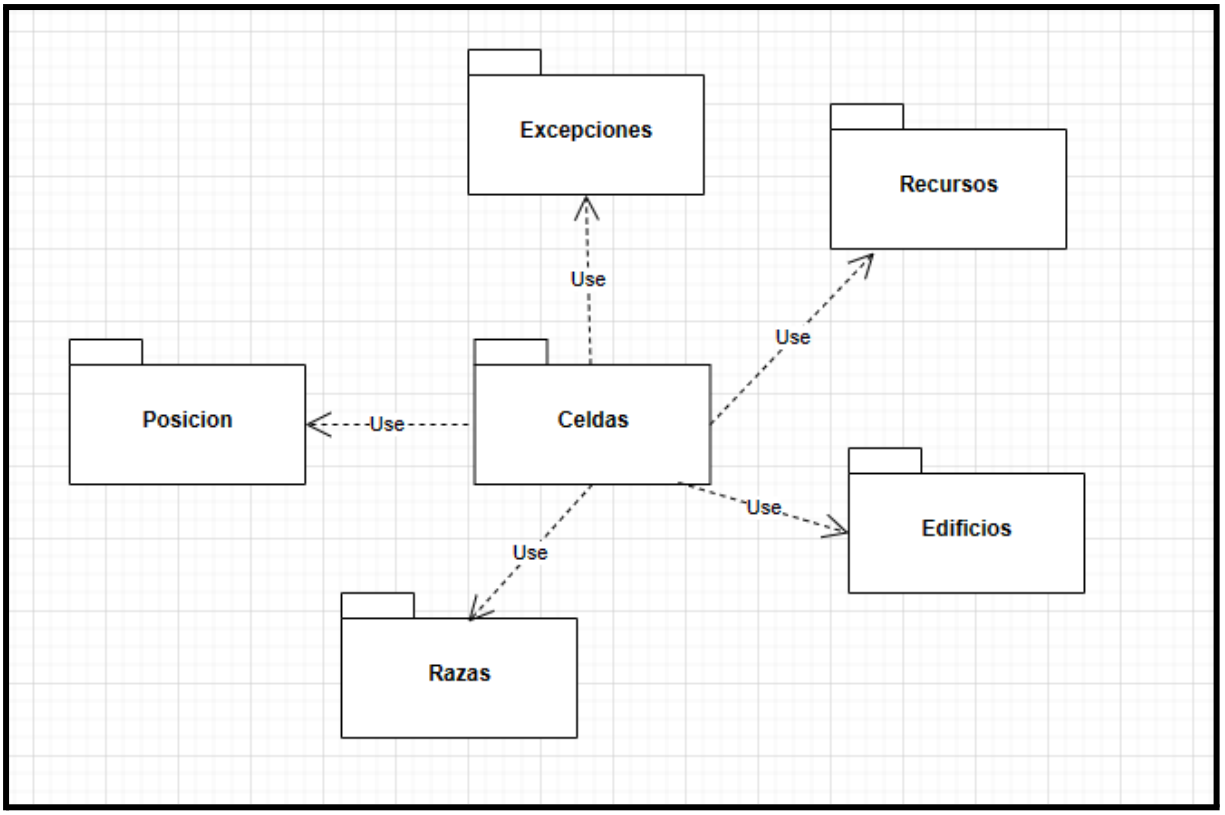


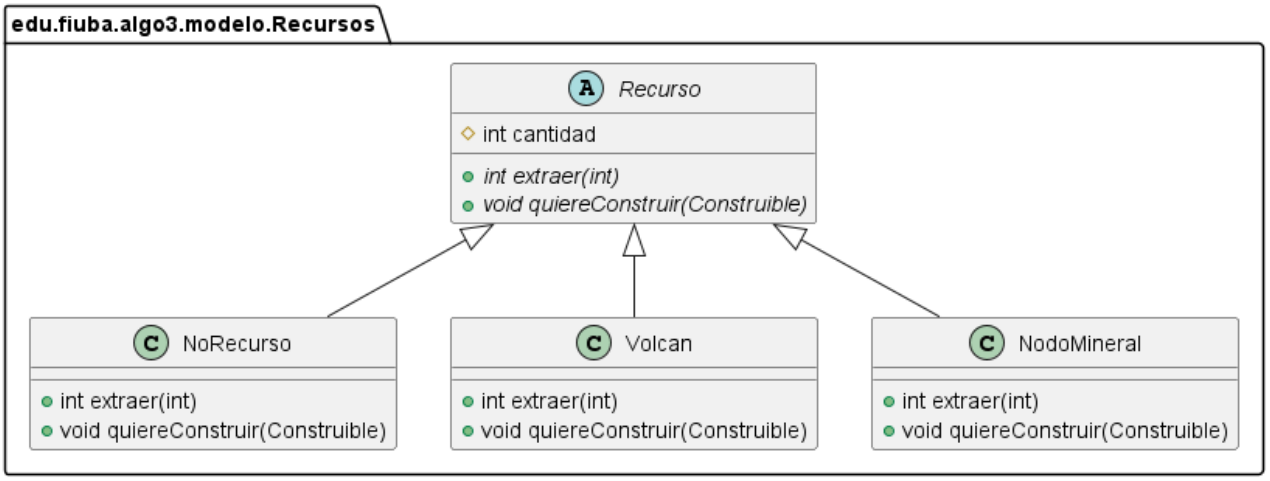
Diagrama de paquetes que hacen funcionar a la celda. Dentro de la propia carpeta Celdas encontramos a la clase Celda junto con las clases que componen su state de tipo de celda.
Luego, requiere una posición desde su carpeta homónima, edificios y tropas que harán de ocupantes (carpeta Edificios y Razas, respectivamente), además de los recursos y ciertas excepciones necesarias.

Recursos

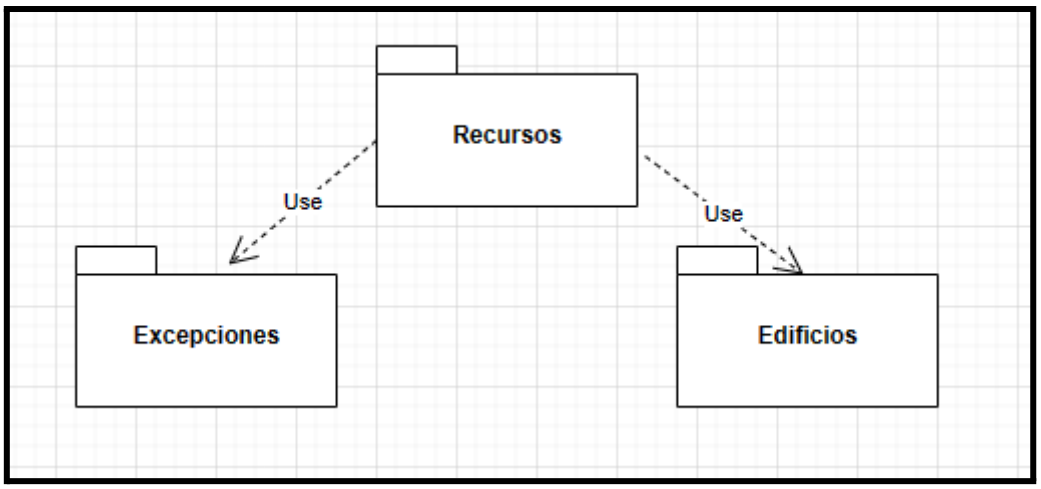
Los recursos constituyen una parte elemental del juego, ya que son los que permiten a los jugadores obtener una moneda de cambio para las creaciones. Tenemos entonces dos tipos de fuentes de recursos distribuidos a lo largo y ancho del mapa: volcanes y nodos minerales. El primero proporciona gas, el segundo, minerales. Lógicamente, las representamos a través de las clases Volcan y NodoMineral, que heredan de la abstracta Recurso.

Siguiendo esta línea, tenemos que la celda puede albergar cualquiera de estos recursos en su atributo, solo hay que instanciar el objeto

correspondiente. Si la celda careciera de recursos, se deberá instanciar un objeto del tipo NoRecurso, una tercera clase hija de Recurso que implementa el patrón null.



Las celdas con recursos admiten el pasaje de tropas y la construcción de edificios recolectores. Una tropa en particular, el zángano, tiene la capacidad de extraer recursos por sí solo. Puede hacerlo en los nodos minerales.



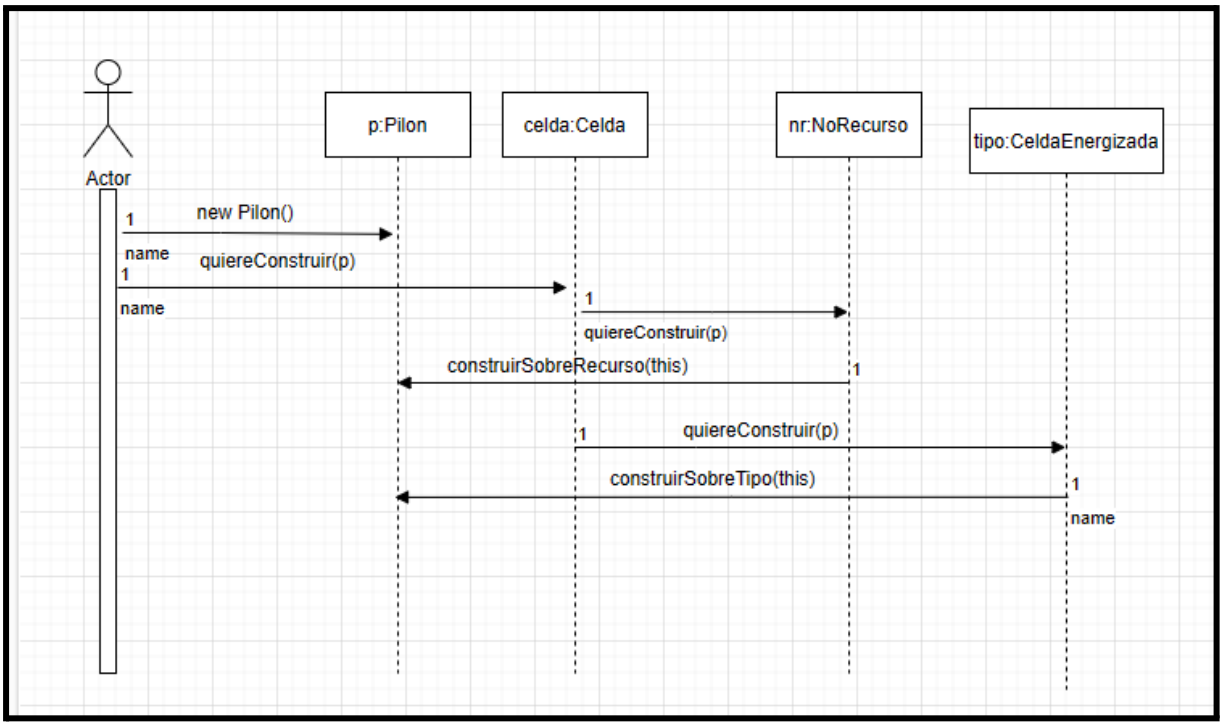
Los recursos requieren la importación de los edificios para validar si estos tienen permitido colocarse encima.

Edificios

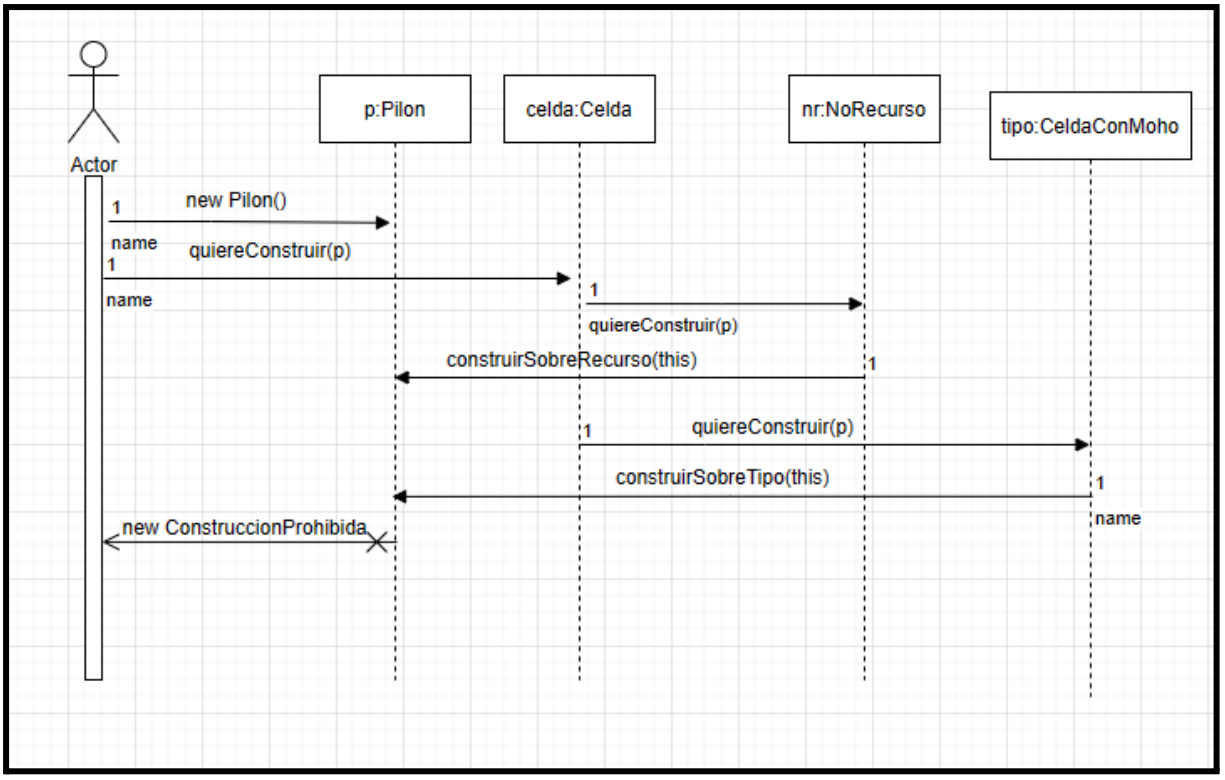
Tienen costo y tiempo de construcción. El beneficio que brindan es variado, así como las condiciones para llevar a cabo su construcción.

Por ejemplo, no es lo mismo construir un extractor en un volcán, que en una celda energizada (el extractor pertenece a la raza Zerg mientras que la celda energizada solo admite edificios Protoss). Para tratar las diferentes combinaciones y requerimientos para construir cada edificio, decidimos utilizar double dispatch.

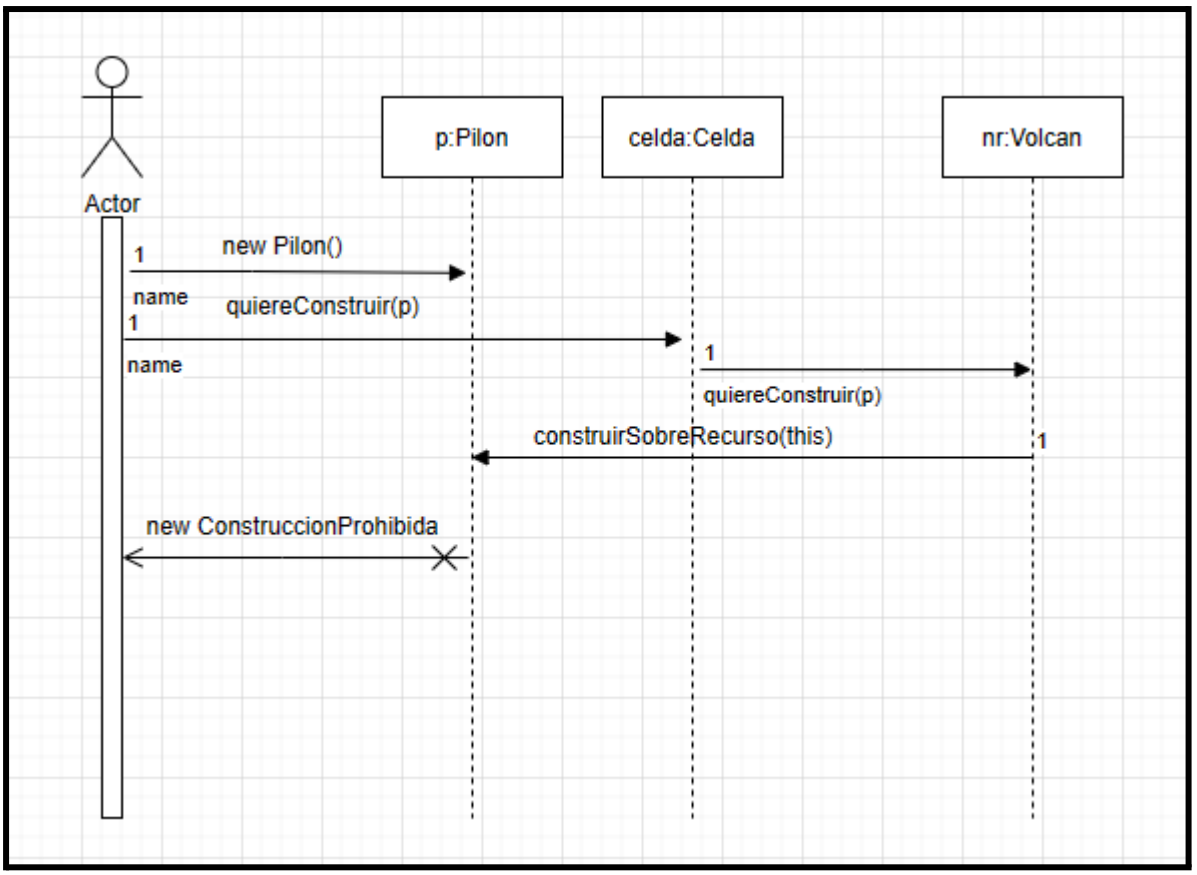
A continuación algunos diagramas de uso.



Ejemplo 1: En este caso la construcción de un pilón es válida, ya que en la celda de colocación se cumplen todos los requisitos: que no haya un recurso generable y que la celda sea del tipo energizada.



Ejemplo 2: Otro intento de construcción de un pilón, pero ahora la celda está infectada con moho, lo cual imposibilita la edificación de los Protoss. Cuando se llama al método construirSobreTipo, el pilón detecta que no puede ser construido en el tipo de celda y lanza una excepción, interrumpiendo el proceso.

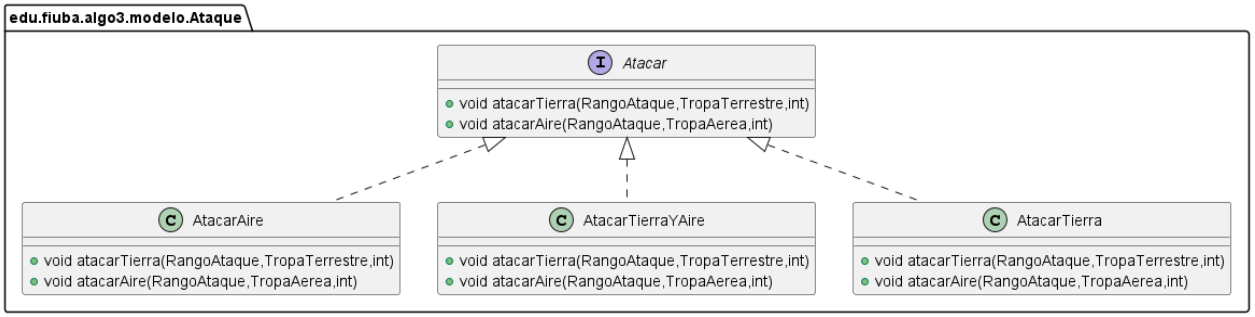


Ejemplo 3: Nuevamente se incumplen las condiciones de construcción de un pilón, ya que en la celda hay un volcán. Recordemos que no debe haber ningún recurso para la construcción de un edificio Protoss (a excepción del asimilador y el nexo mineral).

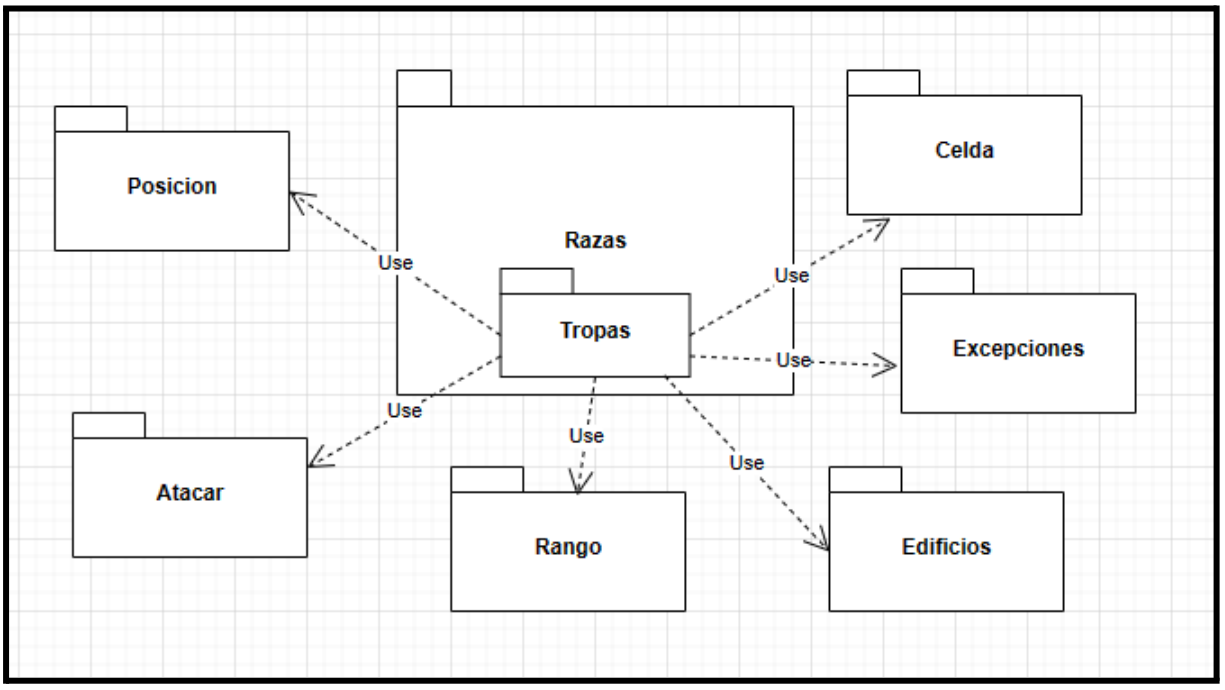
Tropas

Son unidades que se enfrentan entre sí y dañan estructuras. Tienen movilidad y capacidad de ataque. El primer eslabón en su representación en código es la clase Tropa, que alberga algunos métodos generales como los movimientos por celdas (hasta tres por turno) y los ataques. Luego vienen las clases hijas TropaTerrestre y TropaAerea, que se diferencian en que una, al moverse, ocupa celdas por tierra, mientras que la otra lo hace por aire. El siguiente nivel son las tropas particulares, que tienen sus propios métodos de ataque, cantidad de vida, rango de ataque, etc. Algunas hasta implementan habilidades únicas, como por ejemplo volverse invisibles.

Ataque: El ataque es un concepto que se debió tratar aparte. Se comprende que las tropas tienen sus propias capacidades ofensivas. Algunas pueden atacar a tierra, mientras que otras solo a aire, o ambas. En este caso decidimos que la mejor opción era trabajar con el patrón Strategy.



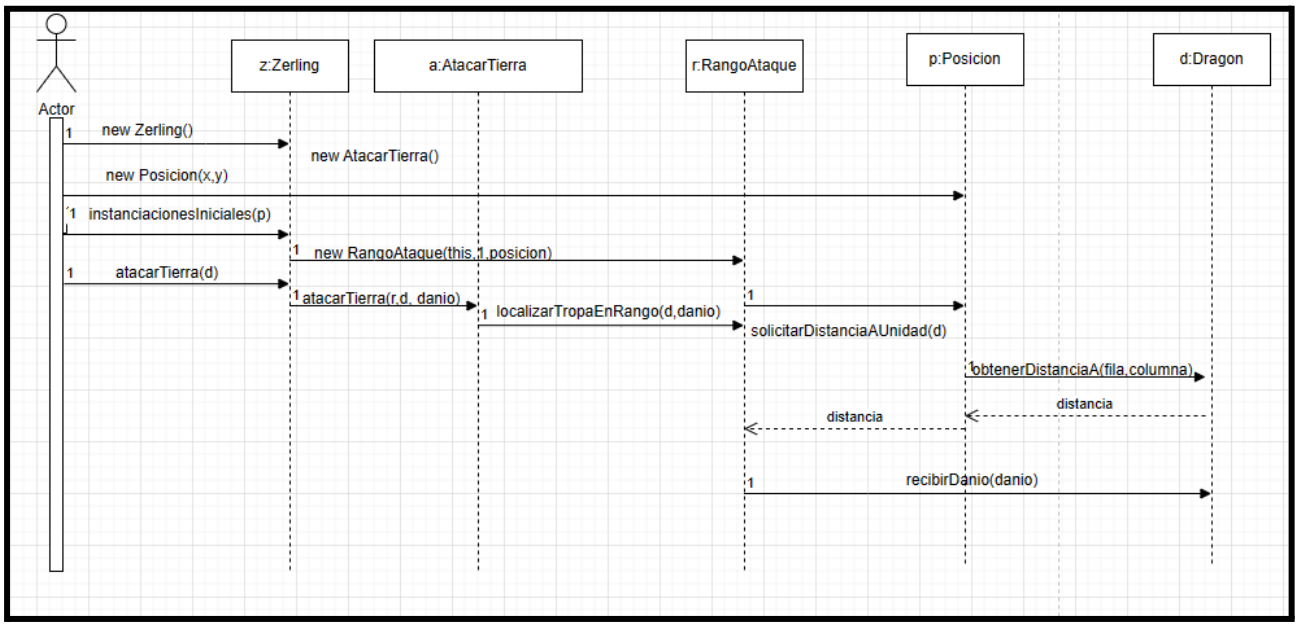
Cada tipo de ataque define los métodos atacarTierra y atacarAire de la interfaz Atacar acorde a sus capacidades. En el caso de la clase AtacarAire, por citar un ejemplo, hará efectivo el método atacarAire (es decir, hará daño al enemigo), mientras que para el método atacarTierra lanzará una excepción notificando que dicho tipo de ataque está fuera de sus capacidades.



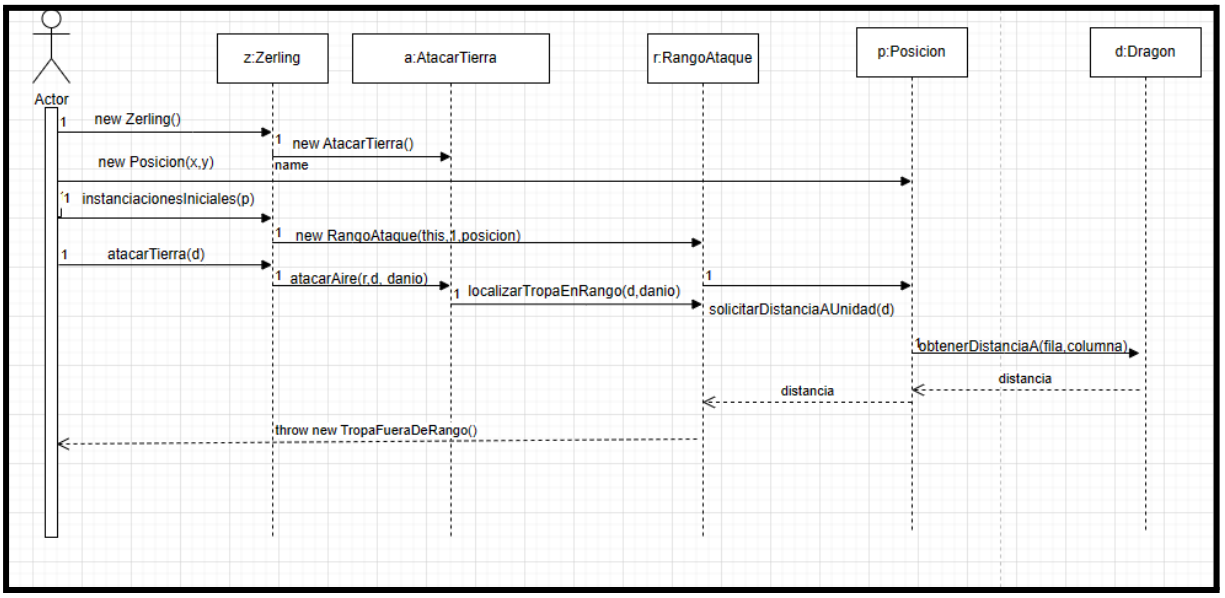
Diagramado de paquetes para las tropas. La carpeta se encuentra dentro de Razas, donde hay algunas clases primordiales como Unidad (la clase abstracta para toda entidad de las razas). Tropas importa

posiciones y celdas donde localizarse, un rango de ataque que le sirve para detectar unidades enemigas, un tipo de ataque (patrón Strategy dentro de la carpeta Atacar), además de edificios a los que puede dañar.

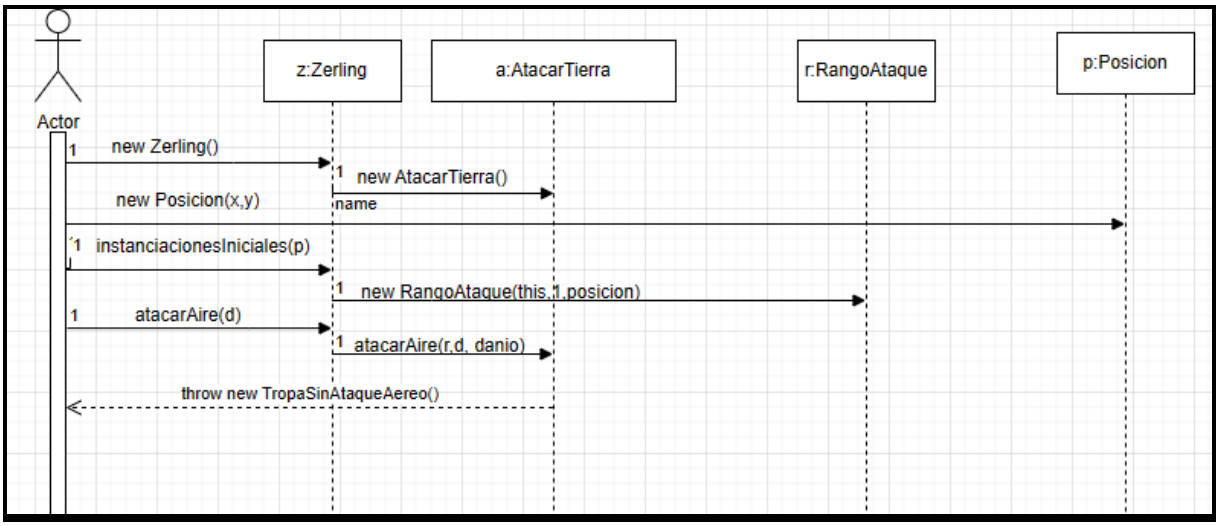
Algunos ejemplos desarrollando la metodología de ataque.



Ejemplo 1: un zerling ataca a un dragón (presupuesto como ya creado) que está dentro de su rango de ataque, por lo que el proceso es exitoso.



Ejemplo 2: Mismos adversarios del ejemplo anterior, pero ahora el dragón está fuera del alcance del Zerling, por lo que el ataque fracasa y se lanza una excepción.



Ejemplo 3: Ahora el proceso cambia ligeramente. Se le solicita a Zerling realizar un ataque aéreo. El problema es que este tipo de tropa no cuenta con dicha capacidad, por lo que la instancia de AtacarTierra se encarga de interrumpir la acción mediante el lanzamiento de una excepción.

Supuestos

A continuación describiremos algunos supuestos que llevamos a cabo según nuestra interpretación.

Reposicionamiento de tropas en celdas ocupadas

-Generación de una tropa en un criadero: Cuando se crea una nueva unidad Zerg, tomamos en cuenta que el criadero de por sí actúa como ocupante, por lo que la nueva tropa no puede ser posicionada en esa misma celda. Para resolver esta situación decidimos explorar las celdas limítrofes hasta hallar alguna vacía. En tal caso la tropa irá a parar allí.

-Colocación de un Zángano en un Extractor: Al igual que en el caso anterior, nos dimos cuenta de que posicionar un zángano sobre un extractor iba a ser un tanto dificultoso ya que el extractor hace de ocupante. Convenimos en que para asignar trabajo a un zángano dentro de este edificio es posible hacerlo tan solo acercando a la tropa a una sola celda de distancia del extractor, ni más ni menos.

Restricción de acciones

-Cantidad de movimientos que una tropa puede realizar: 3 por turno, en cualquier dirección.
-Cantidad de ataques que una tropa puede ejecutar: 5 por turno. En general consideramos que los ataques son relativamente débiles respecto a la vida de los edificios, por lo que decidimos permitir hasta cinco repeticiones de ataques de parte de una misma tropa.

Decisiones propias

Este apartado de texto, si bien no se compone de supuestos, se propone explicar algunos detalles que decidimos plantear a nuestra manera, contrario a lo que solicitaba la consigna.

-Expansión inicial del moho del criadero: el enunciado exigía un radio de cinco, pero evaluando la jugabilidad nos decidimos a cambiarlo a dos, ya que un radio tan grande como el original daba en nuestra opinión, una rápida ventaja a la raza Zerg. En el mapa de 10x10 al menos, suponía una toma completa del mismo en unos pocos turnos.

Fin de partida

Se concreta cuando alguno de los dos bandos se queda sin unidades, es decir, no “existe” más. En tal caso el juego se interrumpe y se anuncia al jugador ganador.

Malas prácticas

Somos conscientes de que en el trabajo pueden presentarse algunas faltas a los principios aprendidos, o implementaciones de código que podrían haberse pensado de mejor manera (un caso muy frecuente que hicimos fue el de comparar clases). En la gran mayoría de los casos esto se debe a que no llegamos a tiempo para emprolijar y corregir ciertos algoritmos que fueron creados inicialmente para que el programa funcione, y desde entonces fueron levemente modificados.