FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Segundo cuatrimestre de 2022

Alumnos:	Suarez Pino, Imanol Tosto, Ezequiel Urquiza, Elias Ibañez, Mateo
Grupo:	7-

1. Introducción

Este informe incluye ciertos diagramas y especificaciones que se tomaron en cuenta al desarrollar el trabajo práctico número dos de algoritmos y programación III. Además, se explica la manera en la que fue encarado el mismo explicitando ciertos supuestos tomados en cuenta al momento de resolver ciertas consignas del mismo.

2. Supuestos

Para empezar con este informe, hablaremos sobre los supuestos que tuvimos al completar el TP. El primer supuesto que tomamos en cuenta fue el tamaño del tablero y su composición, como no había especificaciones en la consigna y tomando en cuenta ciertos comportamientos llegamos a la siguiente idea. El tablero funciona como una matriz de dos dimensiones sin posiciones negativas, los números tanto en la dirección de X, como de Y se expanden del 0 al 50. Estas dimensiones se decidieron debido a la expansión del moho, que era algo no menor, porque se expandía en un espacio por cada dos turnos. Es por eso que las dimensiones son tan grandes, sino el juego se terminaría más rápido. Un segundo supuesto fue que los radios de expansión se medían en distancias hacia la derecha/izquierda o arriba/abajo y no en diagonal, es decir una posición que está en diagonal arriba y a la derecha de un criadero está a distancia 1 y no a distancia 2. Otro supuesto tomado en cuenta fue el tema del terreno en general, por ejemplo, los pisos o posiciones dentro de la matriz. Las posiciones que hay no son espacios ocupados dentro de una matriz, sino que son listas de instancias de distintos objetos como unidades o construcciones, que luego se exponen de manera visual dentro de una matriz. Los espacios libres son tomados como un tipo de piso que fue denominado “tile vacía”. Esta clase tiene muchas instancias y cubren todos los lugares en los que no hay nada. El enfoque que queremos dar con esto, es que un piso tiene o una tile vacía o una unidad o etc, pero algo tiene que tener.

El concepto del vacío fue discutido, y llegamos a la idea de crear un “hueco” de vacío en el medio del tablero, y que no se genere de manera aleatoria, ni que hayan numerosos espacios con vacío. Uno de los supuestos con el tema de los turnos fue que cada vez que haya un usuario jugando tiene la capacidad de tener turnos “infinitos”, es decir que un usuario puede construir lo que quiera hasta que decida que termina su turno. Ocurre algo parecido con el tema de los movimientos, asumimos que las unidades se podían mover de manera infinita en los lugares correspondientes, de manera similar ocurre con los ataques una unidad puede atacar infinitas veces no hay límite. Si bien esto rompe un poco con la jugabilidad, no nos parece mal dejarlo para futuras implementaciones o como una mejora de lo que tenemos. Otra de las suposiciones relacionadas a las unidades es que se acepta tanto el fuego enemigo como el amigo, es decir que una unidad zerg puede dañar a otra zerg y lo mismo con los protoss. Finalmente, un supuesto relacionado a los zerg tiene que ver con el tema de la extracción de recursos, porque un extractor puede extraer gas, pero no tenemos quien extrae cristales. Supusimos que el encargado de sacar cristales es el zángano que en nuestro caso no es una estructura como tal sino que es una unidad y se comporta como una unidad, pero tiene la capacidad de aportar a la economía de su raza extrayendo cristales, decidimos que el zángano con solo estar a radio uno del cristal pueda extraer mineral de este.

3. Detalles de implementación

Los ataques pueden ser del tipo aéreos o terrestres, en base a eso se crearon dos clases “Aire” y “Tierra”. Cuando una unidad ataca esta puede atacar por tierra o por aire, pero la realización del ataque depende de si el objetivo es terrestre o aéreo, pues un ataque aéreo a un objetivo terrestre no es posible. Por lo que se recurre a un método de comparación que viene dado en una interfaz “Ataque” la cual implementan las clases Aire y Tierra. De modo que los ataques saben compararse entre sí y pueden

decirnos si matchean el tipo de ataque con el tipo de superficie del atacado. O sea que se usa un poco de “tell don’t ask”, aunque es en las propias clases y no desde afuera (pues se comparan entre ellas y no hay un tercero en el medio), a cambio de no implementar un “double dispatch” el cual a futuro violaría Open/Close.

Estamos contentos con la implementación de la economía de las dos razas, pues esta está delegada en una clase especial la cual maneja los recursos (llamada “Economía”). Esta es usada por cada clase que tenga algún tipo de dependencia con los recursos, por ejemplo, diciéndole que se reste cien de mineral por que el costo de crearse ellos es ese, o si es un nexo mineral le dice economía agregate 10 de mineral porque lo acabo de extraer y así hay más escenarios. En ningún momento se rompe ni se viola algún principio o buena práctica, tampoco supuso una traba para el desarrollo del TP y a medida que se sumaban nuevos requisitos esta clase no fue cambiada en absoluto. Aclaración: hay una instancia de economía para Zerg y otra para Protoss.

En el TP hay una clase llamada “Posición”, la cual es la encargada de administrar las posiciones. Cada construcción y unidad tiene una posición única, que solo puede llegar a repetirse con los pisos, por ejemplo: en la posición de una guarida hay a la vez moho, o también en donde hay un acceso de los protoss debajo de él sabemos que hay un piso de energía. A lo largo del TP se usa mucho un método “Equals” el cual permite la comparación de posiciones, pero al igual que el ataque antes mencionado la comparación se mantiene dentro de la clase posición no interfiere un tercero. Lo negativo es que al igual que el ataque hay un incumplimiento de “Tell don’t ask”, pero es la única forma de comparar posiciones y encima que el comportamiento esté encapsulado y ninguna clase que dependa de posición sepa como esta se maneja. Es muy útil desde la vista esta clase porque la vista no sabe nada del modelo, solo sabe en qué posición esta clickeando, después el modelo se encarga de hacer lo necesario recibiéndola por parámetro.

El modelo maneja a las clases en listas, por ejemplo, unas listas con edificios zergs, otra con protoss, otra con volcanes, etc. Esto permite que el modelo esté completamente aislado de la matriz en la cual después todo va a parar. Esto se evidencia en que en el modelo no hay una matriz si no que está recién aparece en la vista, porque nosotros decidimos usar una matriz.

En la vista se hace uso de un “Observer” que no es tal cual al patrón observer, pero se parece. Este notifica a los que lo escuchan para que los cambios se lleven a cabo cuando al interactuar con el modelo este no tira excepción, si no que lleva a cabo la acción pedida. Entonces cuando esto ocurre se refresca la pantalla volviéndose a crear, teniendo consigo el cambio producido. Es como un update fuerte en el sentido de que es bien general. O sea que es un observer, pero por fuera del modelo.

El “Manager” es un “information expert” sobre las construcciones y unidades, en el sentido que el tiene a todos, y con su conocimiento delega en los demás y de esa forma se llevan a cabo las acciones. La interacción con el modelo es casi toda a través de él. Existen dos mini manager, “FloorManager” y “UnidadManager” en los cuales el manager delega, pero bastante, como por ejemplo en floor manager todo el manejo de pisos (Decidir las prioridades de los pisos, expansión del moho, energización, etc.) y en unidades el manejo de las unidades (Que se puedan mover, que puedan atacar, chequean la evolución).

Las construcciones y unidades no se crean en el modelo si no que vienen creadas desde afuera y en el modelo se las válida, esto permite ahorrar tiempo y hacer las cosas de forma mucho más general, ya que pongo un método que reciba una construcción zerg por parámetro y listo me sirve para cada construcción zerg y no tengo que tener un método por tipo de construcción

Para la construcción de las unidades se elige un radio de uno alrededor del edificio que construye dicha unidad. Si este rango está todo ocupado la unidad no se creará, pero se pierden los costos de creación. (No hicimos el caso de que si está lleno ese radio que no se pueda seguir construyendo)

Aplicamos el patrón MVC donde tenemos la vista separada del modelo (El modelo no realiza ninguna

acción de la vista) y un controlador que da inicio a las pantallas y a la primera vista.

Otro detalle a aclarar es que cuando un recurso se agota, si bien sigue estando el edificio que lo extrae y el mineral sigue estando en el mapa, ya no se extrae nada de allí. No hay un aviso al usuario, pero ese es el funcionamiento.

4. Diagramas de Clase

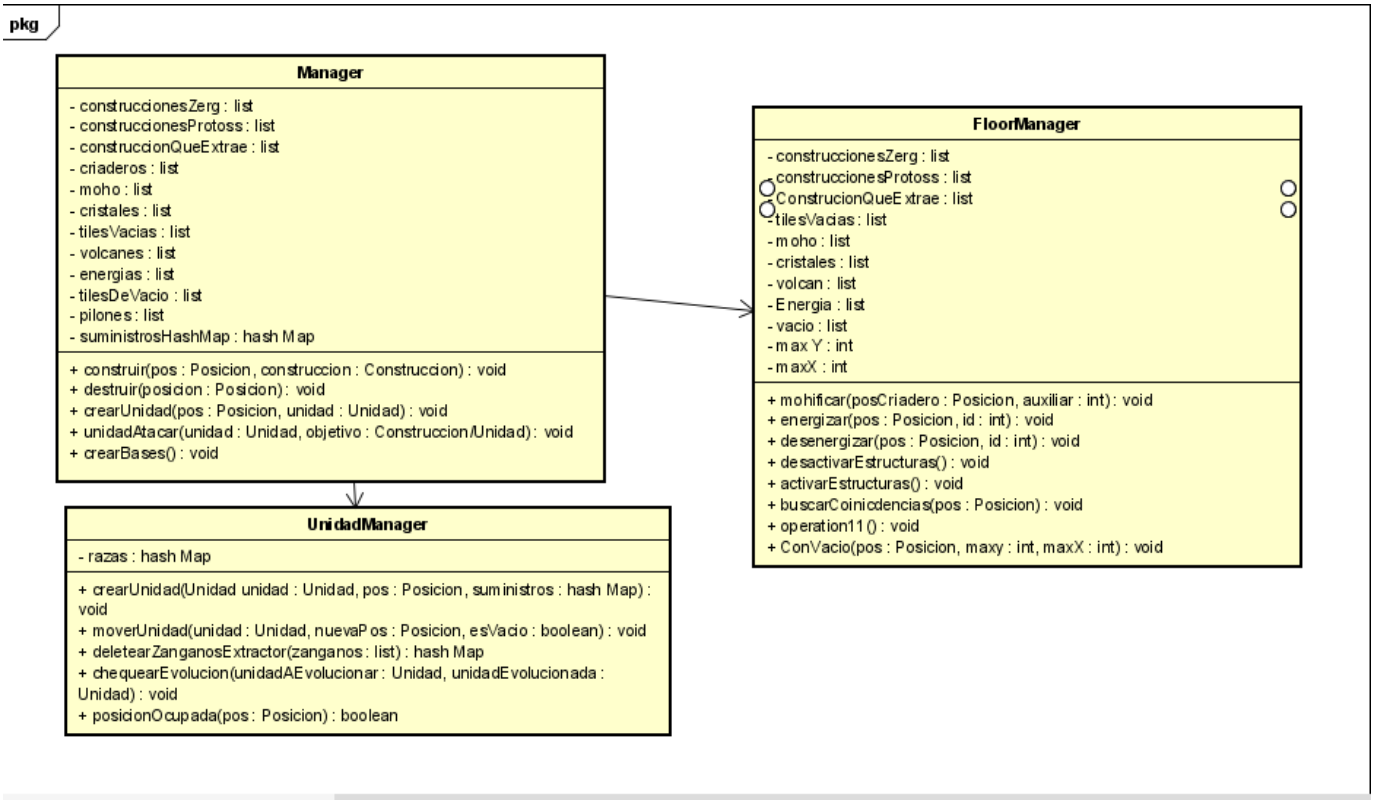

Figura 1: Diagrama de clases 1.

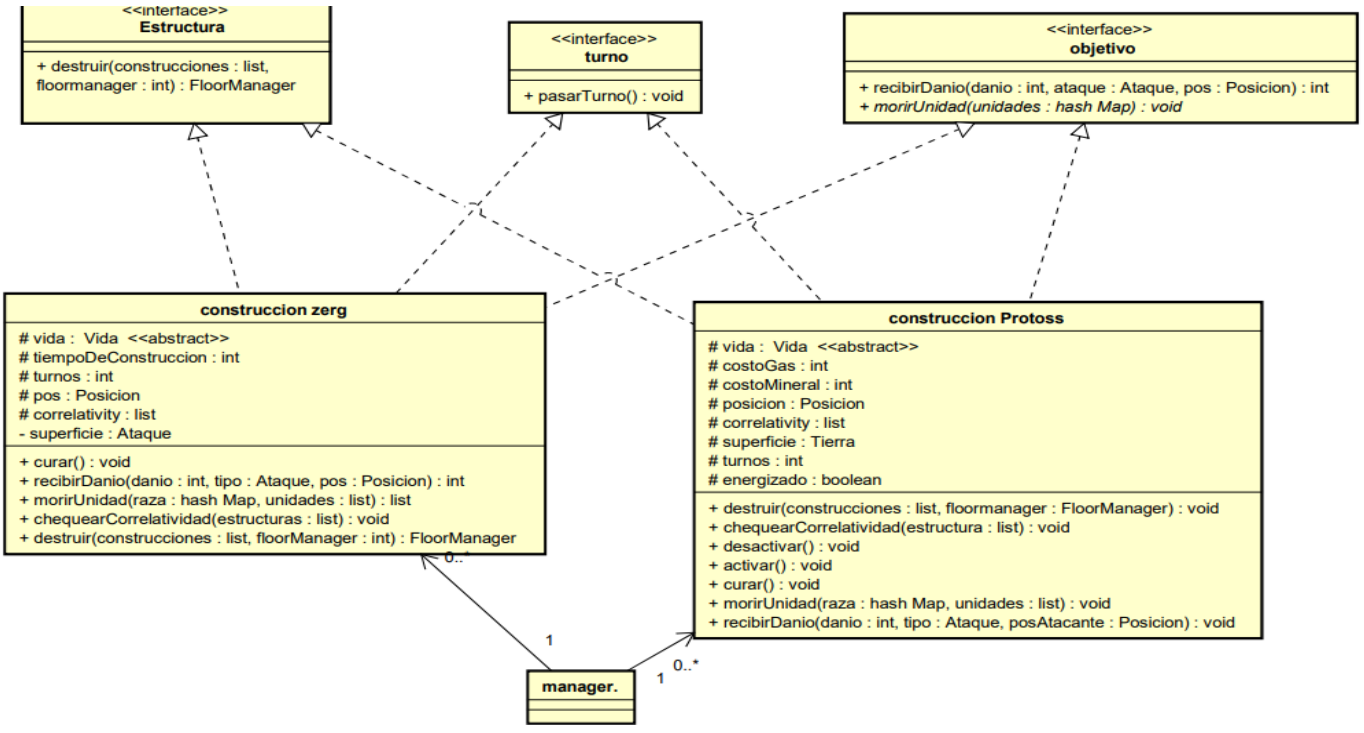

Figura 2: Diagrama de clases 2.

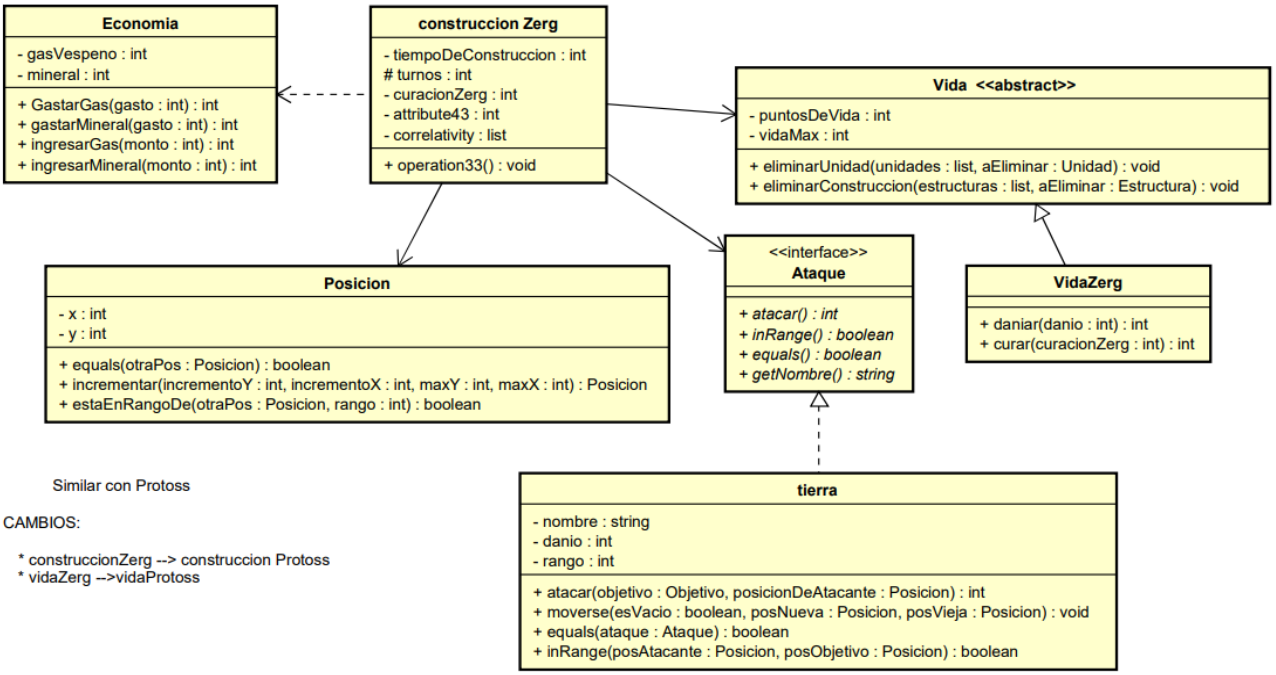

Figura 3: Diagrama de clases 3.

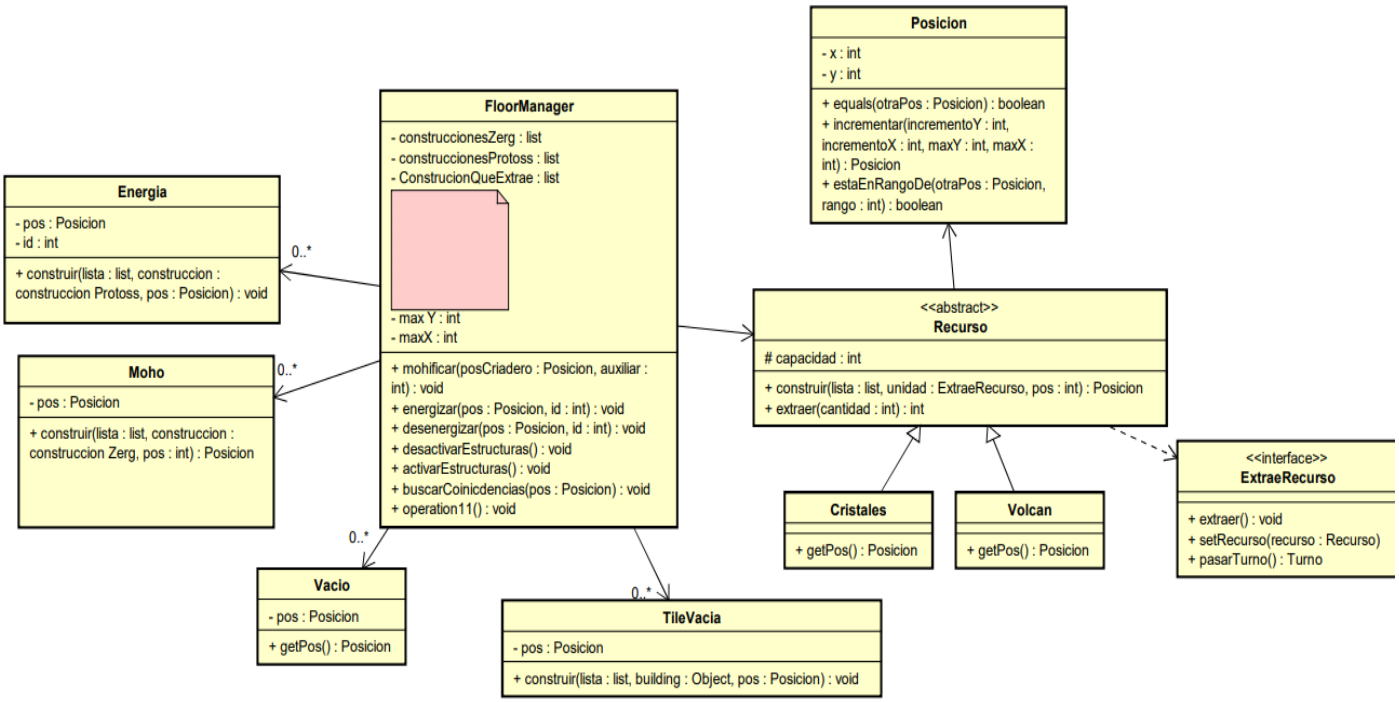

Figura 4: Diagrama de clases 4.

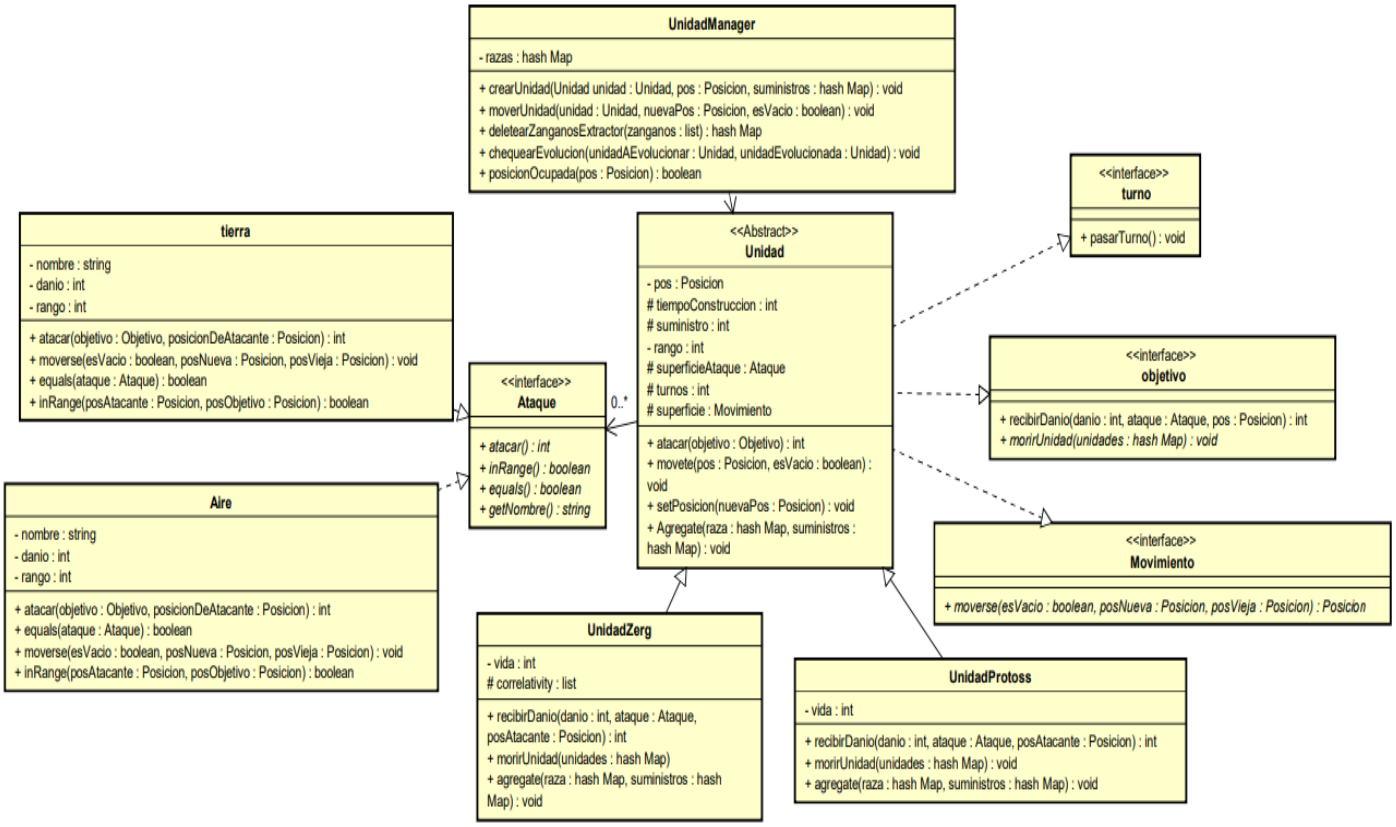

Figura 5: Diagrama de clases 5.

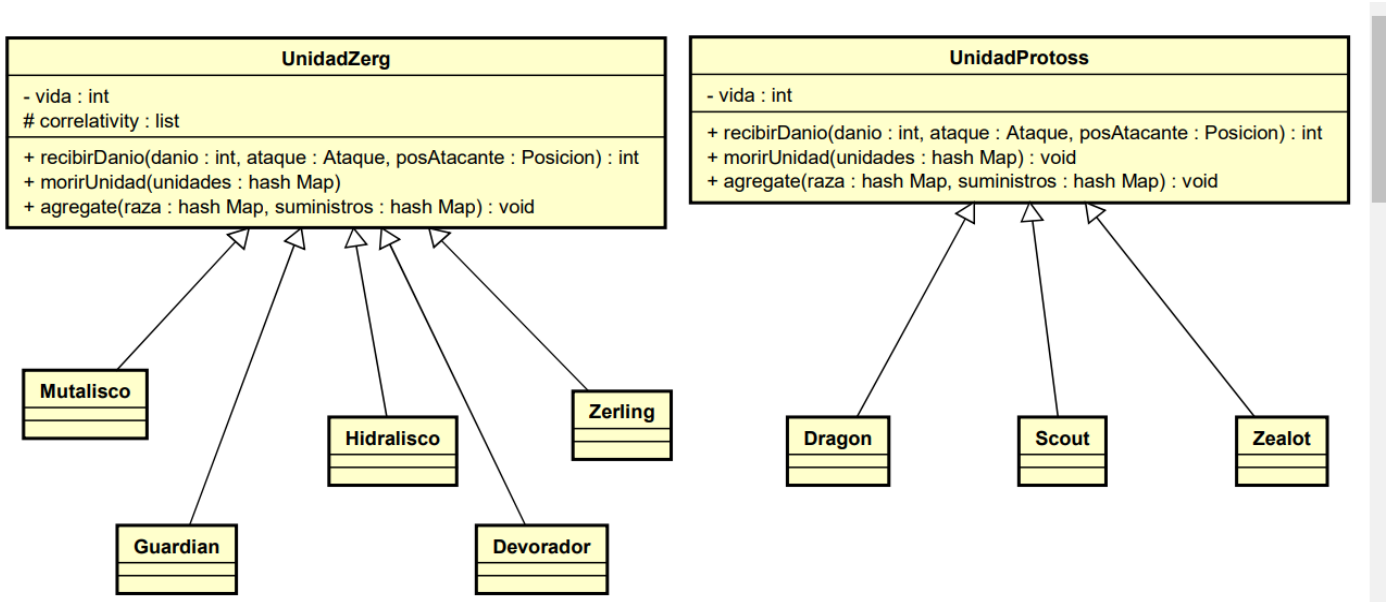

Figura 6: Diagrama de clases 6.

5. Diagramas de secuencia

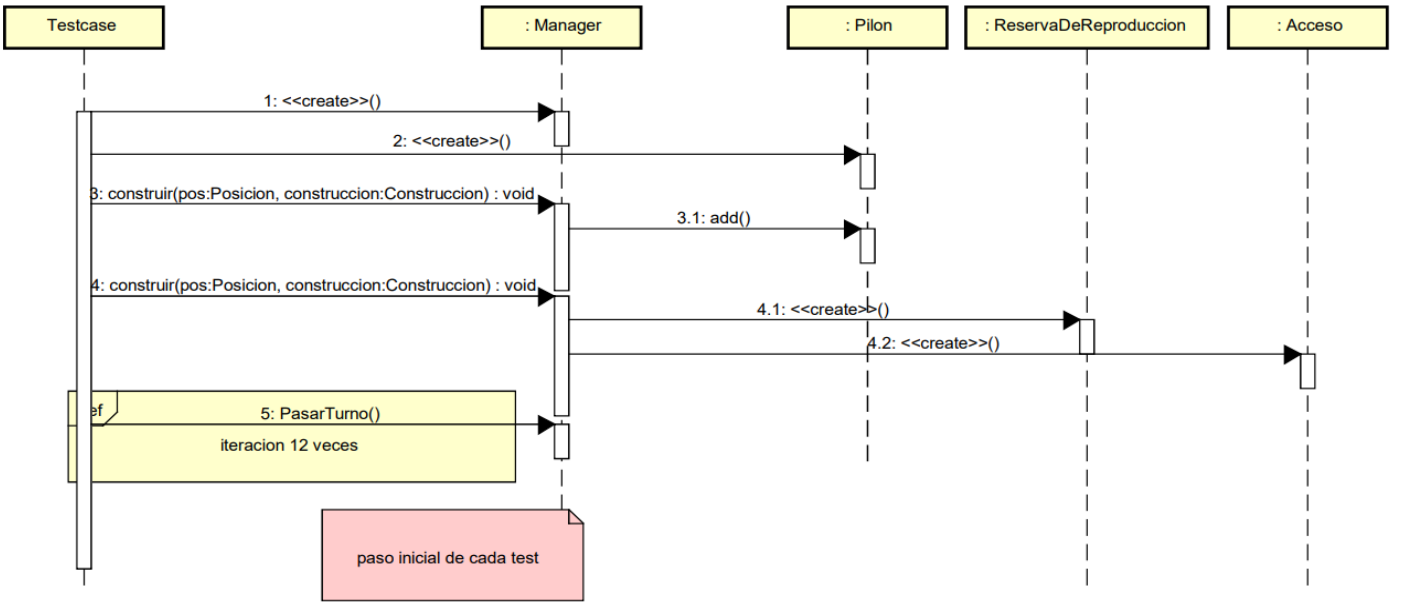

Figura 7: Diagrama de secuencia inicialización general.

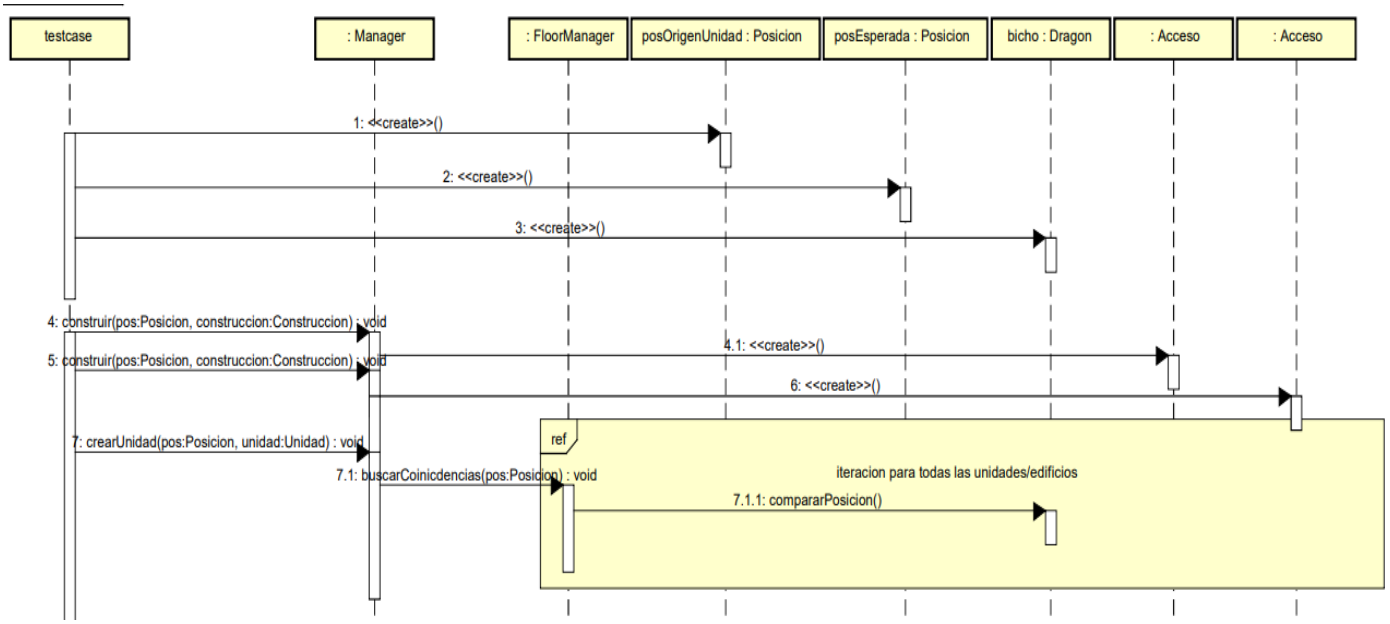

Figura 8: Diagrama de secuencia caso de uso creación de unidad en el manager.

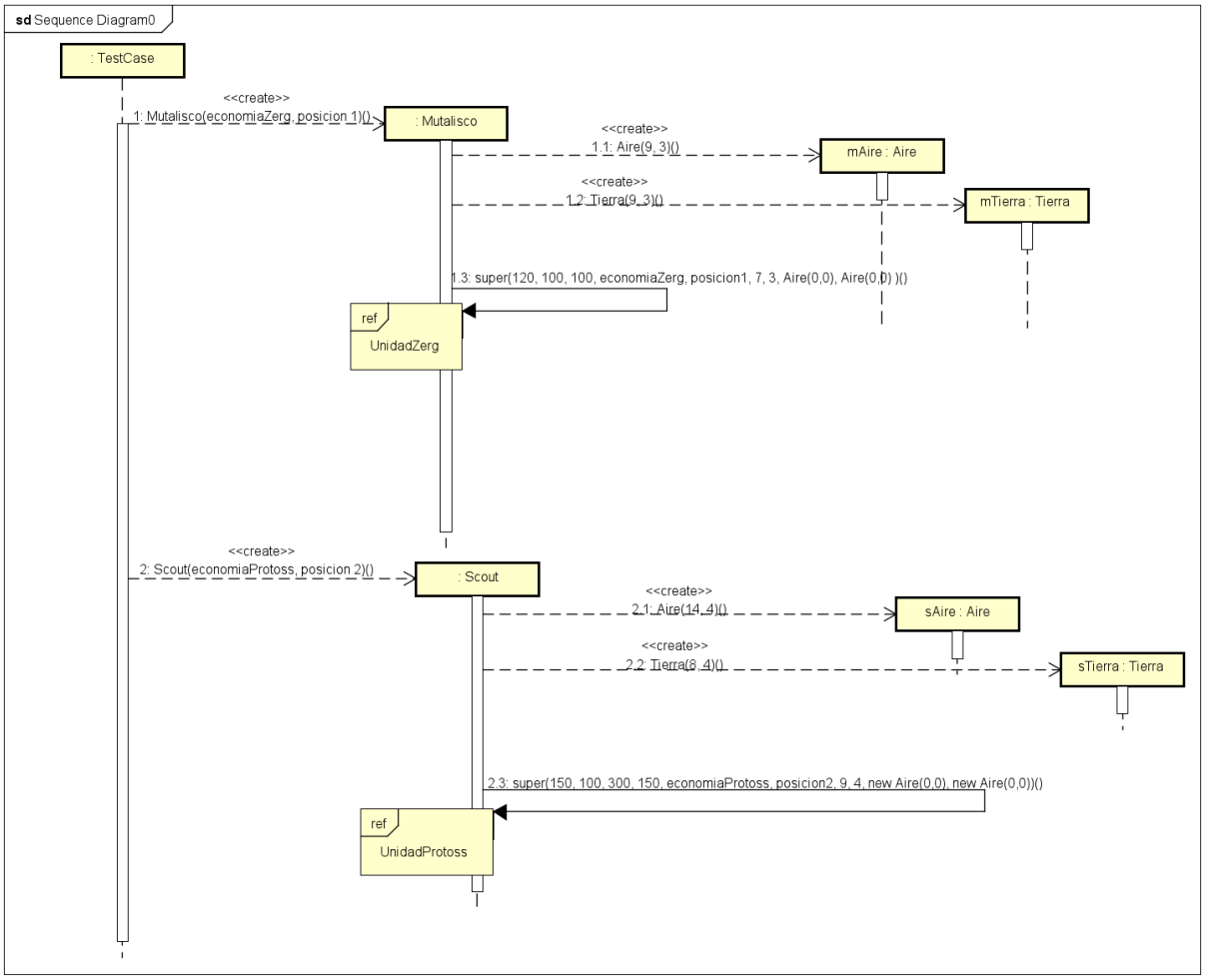

Figura 9: Diagrama de secuencia caso de uso Ataque de una unidad a otra unidad o construcción. En concreto se aprecia la creación de las dos unidades.

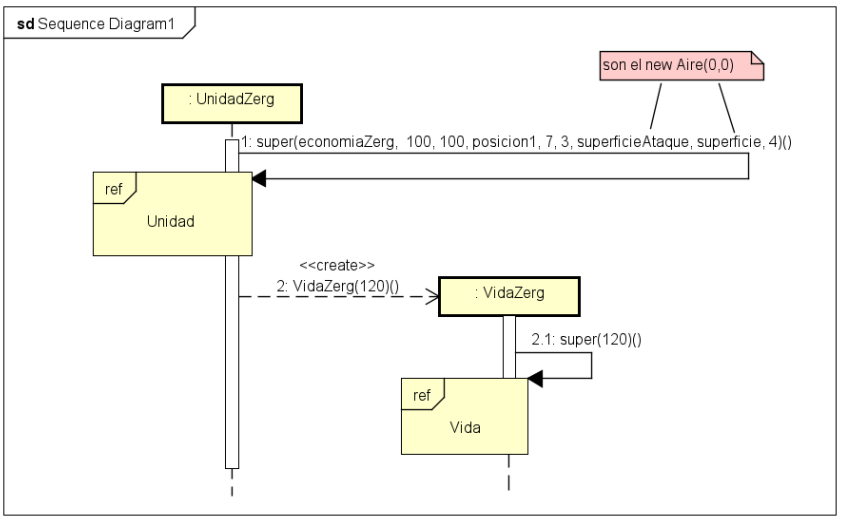

Figura 10: Diagrama de secuencia creación de una unidad zerg (continuación de la creación del mutalisco) .

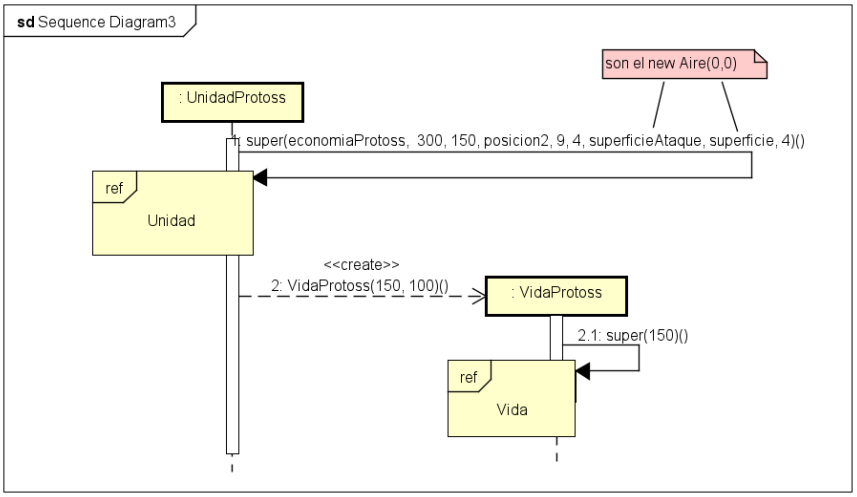

Figura 11: Diagrama de secuencia creación de una unidad protoss (Continuación de la creación del Scout).

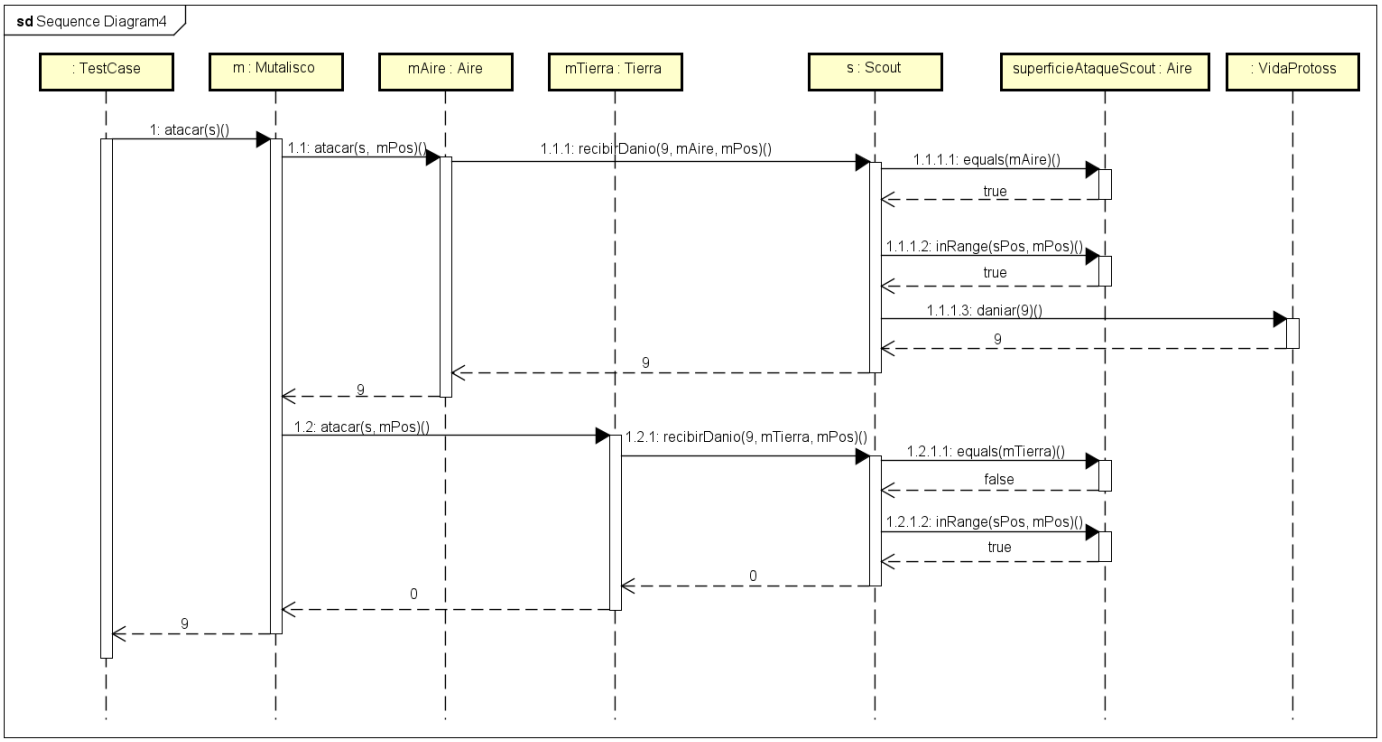

Figura 12: Diagrama de secuencia de unidad zerg ataca a una unidad protoss (Culminación de los diagramas del 9 al 12).

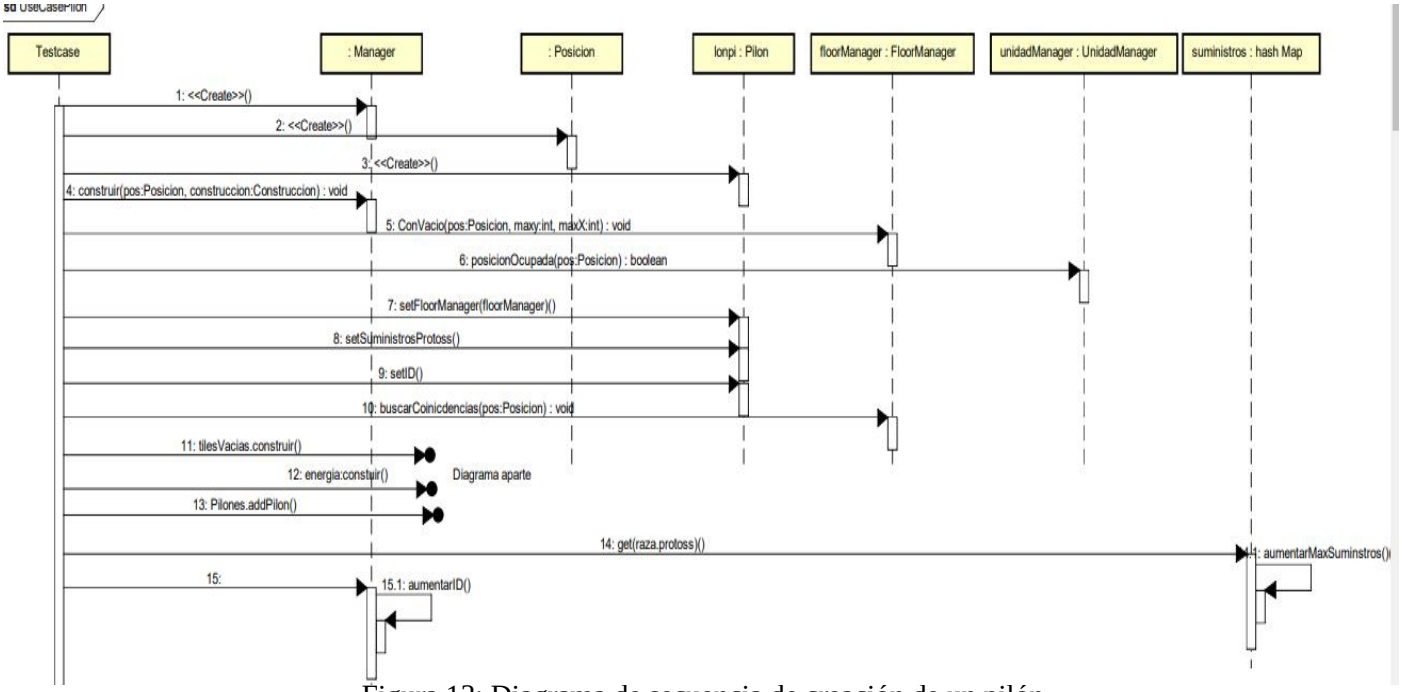

Figura 13: Diagrama de secuencia de creación de un pilón.

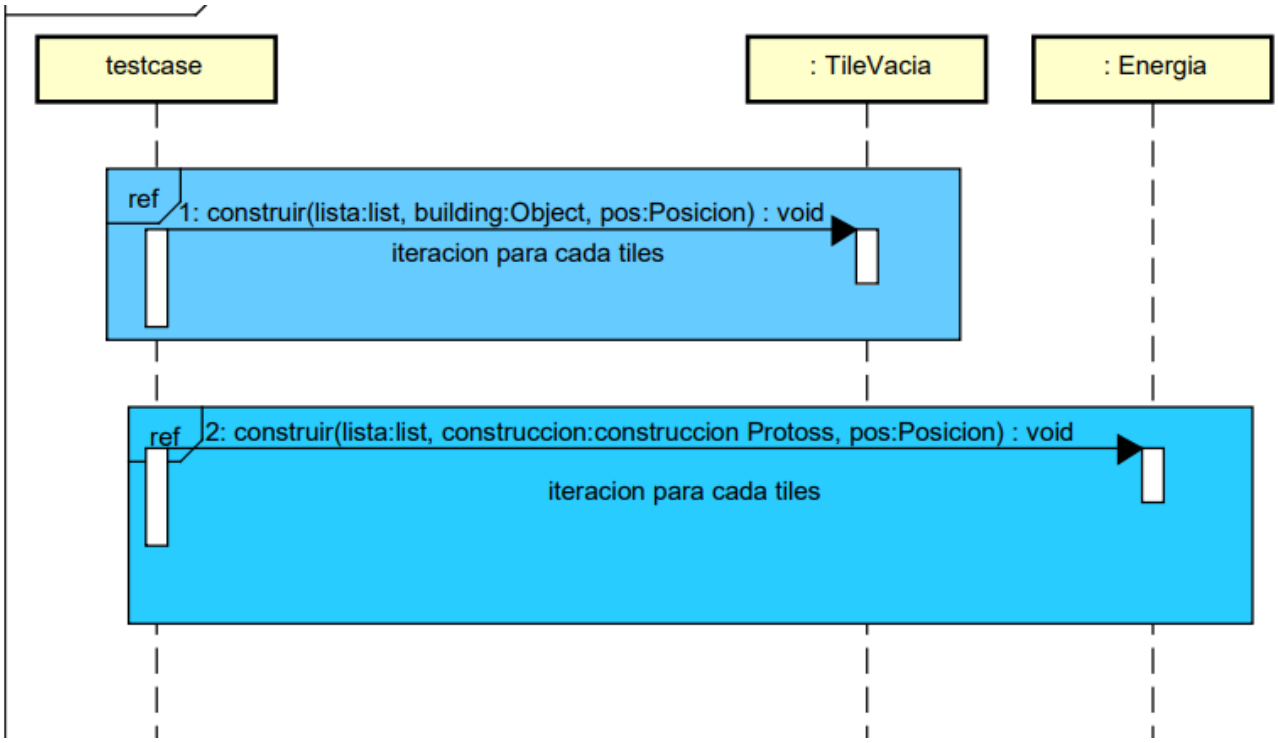

Figura 14: Diagrama de secuencia que muestra la parte de Creación de un pilón en la que se indica que se hace en otro diagrama.