



Trabajo Práctico 2 — AlgoStar

[7507/9502] Algoritmos y Programación III

Grupo 3

Segundo cuatrimestre de 2022

Alumno	Padrón	gitHub
Camila Gonzalez	105661	c-gonzalez-a
Pilar Paz Blanco	105600	ppazb
Mateo Cabrera	108118	m-cabrerar

Índice

1. Introducción	2
2. Supuestos	2
3. Modelo de dominio	2
4. Diagramas de clase	3
5. Detalles de implementación	4
5.1. Implementación de Juego	4
5.2. Implementacion de Mapa	5
5.2.1. Casilleros	6
5.2.2. TipoCasillero	7
5.3. Implementacion de Jugador	8
5.4. Implementación de Inventario	9
5.5. Implementacion de Unidades	9
5.5.1. Unidades Moviles	10
5.5.2. Edificios	12
5.5.3. Vida, Escudo, Danio y Superficie	14
6. Excepciones	15
7. Diagramas de secuencia	17
7.1. Expandir Moho al pasar turno	17
7.2. Creacion de un Juego Nuevo, registro un jugador y creo un mapa listo para usarse	18
7.3. Una Unidad Movil ataca un edificio y lo rompe	19
7.4. Creo un Edificio Zerg sobre un Casillero energizado	20
7.5. Dos unidades pelean hasta que una muere	21

1. Introducción

El presente informe reune la documentación de la solución del primer trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar el Juego AlgoStar en Java utilizando los conceptos del paradigma de la orientación a objetos vistos hasta ahora en el curso.

2. Supuestos

- Los edificios Zerg, y los escudos Protoss recuperan uno de vida por turno.
- Los edificios en construcción se destruyen inmediatamente reciben cualquier tipo de daño.
- El Nexo Mineral extrae veinte unidades de Mineral por turno
- Los Zerg inician con un criadero, y los Protoss con un Pilón.
- El amo supremo se puede construir desde el principio de la partida.
- Los suministros no pueden ser negativos. El mínimo numero es cero.
- La correlatividad de edificios es habilitante, es decir, una vez construido un edificio, se puede construir su correlativa, independientemente de si luego el primero es destruido.
- Diferenciamos suministros empleados de suministros disponibles.
- El juego se gana cuando el contrincante no tiene edificios. Notese que esto es independiente de si tiene o no Unidades Móviles restantes.

3. Modelo de dominio

El presente trabajo busca diseñar el juego AlgoStar, basado en StarCraft, priorizando el código por sobre la representación visual. Para la elaboración del mismo se buscó cumplir con los principios SOLID y seguir los pilares de programación en objetos.

El juego consta de dos jugadores, cada cual asociado con una raza alienigena, que tienen el objetivo de destruir toda construcción del enemigo. Las reglas y metodología de juego fueron modeladas en base a las indicaciones dadas por la cátedra, los casos de uso y los supuestos mencionados anteriormente.

Buscaremos explicar el modelo iniciando desde una visión general avanzando a sus partes más específicas de forma progresiva. Se recomienda primero mirar el diagrama de clases general de la sección siguiente con detenimiento para luego poder entender los detalles de implementación de forma cumulativa.

4. Diagramas de clase

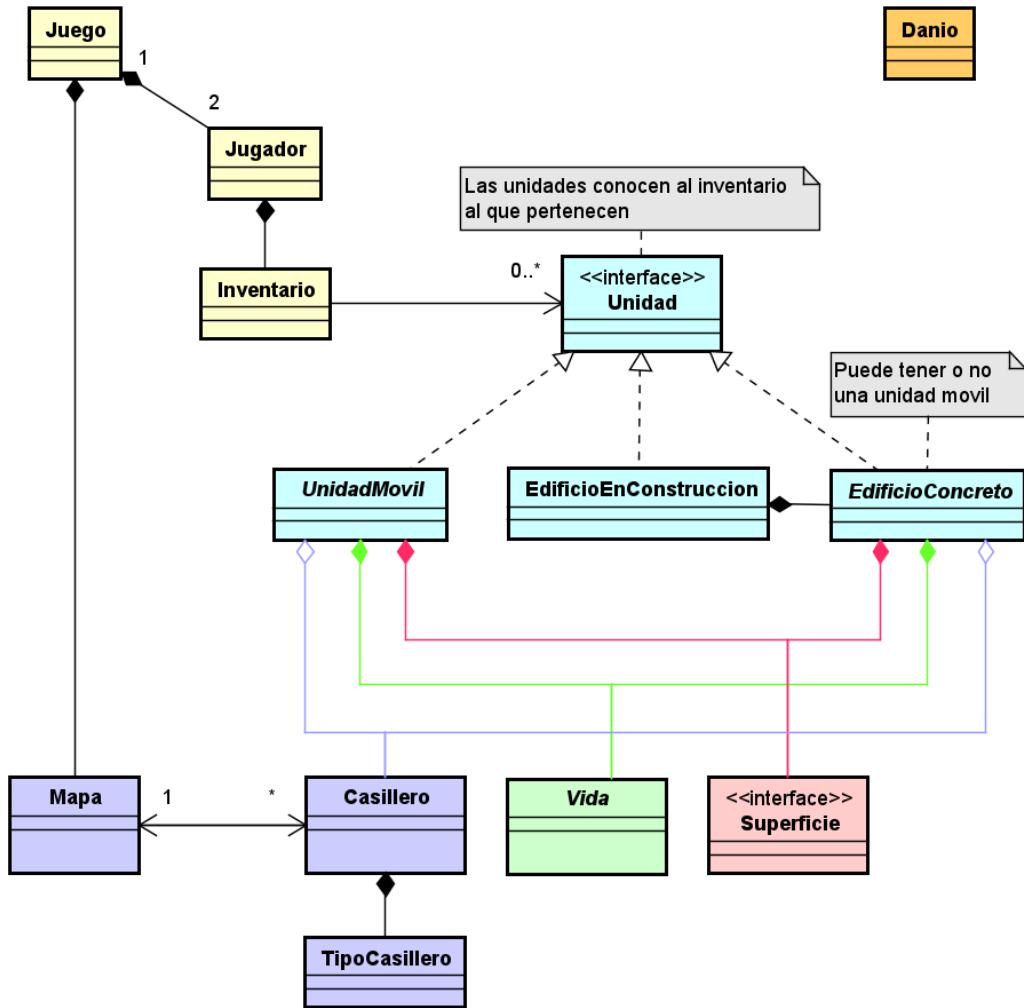


Figura 1: Diagrama general del modelo

5. Detalles de implementación

5.1. Implementación de Juego

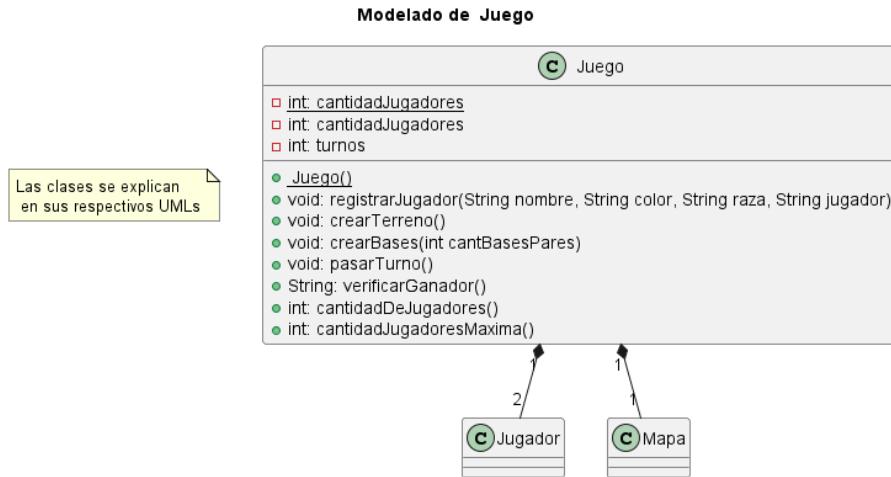


Figura 2: Diagrama de la clase Juego

La clase Juego busca englobar las demás clases participantes y crear los elementos necesarios para iniciar una partida. En la función constructora Juego() se crea el mapa, se registran los jugadores y se inicializa en turno 0 el juego. El mapa como tal solo tiene casilleros vacíos y hay que determinar un terreno y colocar bases, es decir, darle variación en forma de casillas de Moho, Espaciales y Nodos. Para darle esa variación se decidió que de forma proporcional a las casillas que tiene el mapa, un porcentaje se vuelven Nodos de Gas, Nodos Minerales, Moho y casilleros Espaciales. En tanto a los casilleros espaciales, debido a que las unidades voladoras aparecen más adelante en el juego, se decidió restringir un poco más su zona de aparición para evitar que estén cerca de las bases. A su vez, de forma randomizada, cuando se crea un casillero Espacial, hay chance de que algunos de sus adyacentes también lo sean, para que se creen pequeñas zonas de casilleros espaciales.

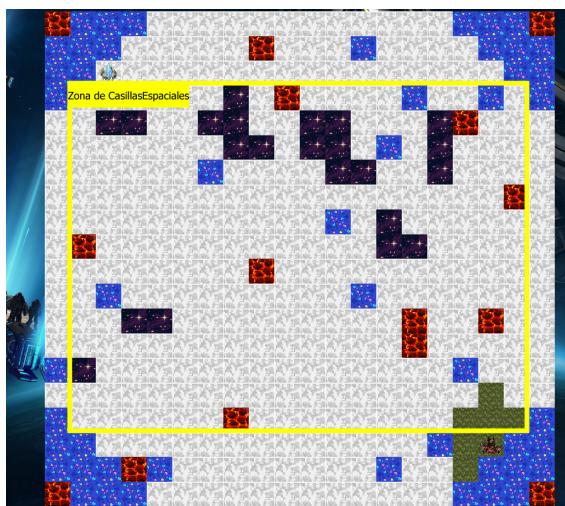


Figura 3: Mapa una vez creado, tras colocar bases y variaciones de casilla. Se pueden ver en pantalla la zona restringida de las casillas espaciales.

El juego, a su vez, puede crearse con 2,4 o 6 bases que aparecen equidistantes en el mapa.



Figura 4: Visualización de Mapa con 2, 4 y 6 bases.

También se tuvo en cuenta que la base que se crea en el extremo inferior, es siempre la del jugador 1 adaptándose siempre a la raza que elija.

Finalmente, Juego, al ser la clase mas general, es la que llama al pasar turno que desencadena los pasar turno individuales de cada jugador (con sus respectivos inventarios) y del mapa.

5.2. Implementacion de Mapa

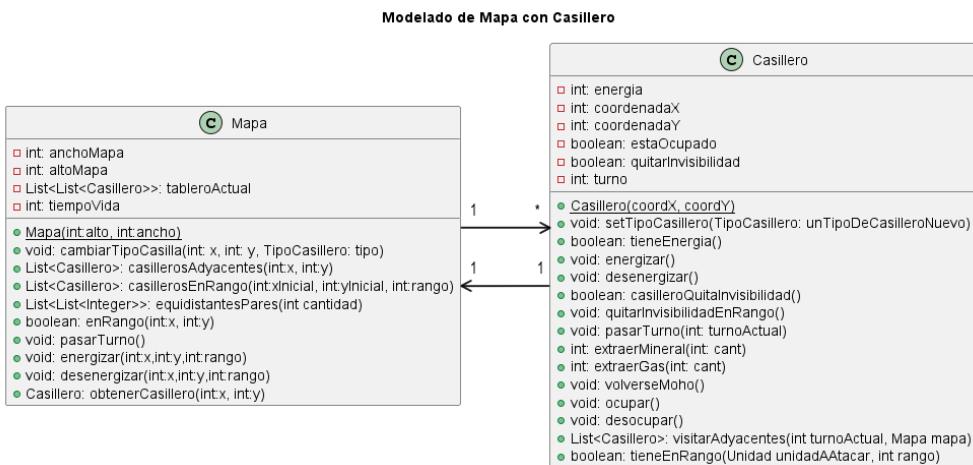


Figura 5: Diagrama de la clase Mapa con sus relaciones mas directas

Para explicar Mapa también debemos referirnos a Casillero, puesto que el mapa es un tablero que consta de muchos Casilleros en una relación muy fuerte. En si el mapa sólo es conocido por sus casilleros y el Juego: consideramos que las Unidades solamente debían saber dónde están ubicadas y el casillero solamente debe saber si está ocupado o no, independientemente de lo que tenga encima, lo cual nos permitió respetar más el encapsulamiento, pero generando una relación mas fuerte entre Mapa y Casillero. Esto no presentó mayores problemas puesto que si bien Casillero es una de las clases que mas chequeos tiene que realizar, el mapa sólo se encarga de hacer tratamientos generales de los casilleros que lo constituyen: un casillero puede pedirle al mapa q energice sus adyacentes o pedirle los casilleros que se encuentren en un rango determinado, pero los que estan encargados de energizarse, cambiar su tipo u otras operaciones es el casillero.

En tanto a PasarTurno se vió la necesidad de un registro de turno interno del mapa, puesto que operaciones como expandir el moho solo ocurren cada cinco turnos. A su vez, como el mapa actualiza una a una sus casillas, si no se tiene considerado si ya se visito o no un casillero en el turno actual hay situaciones donde el funcionamiento no seria el esperado. Por ejemplo, si un moho se expande a una casilla que todavia no paso de turno, y luego se visita esa casilla, no corresponde q el moho se vuelva a expandir.

5.2.1. Casilleros

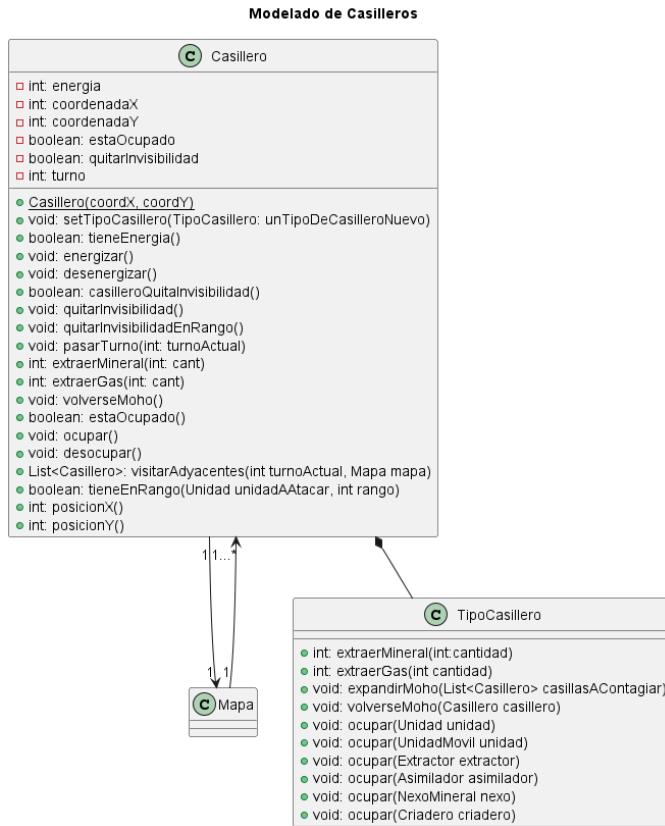


Figura 6: Diagrama de la clase Casillero con sus relaciones mas directas

Como ya mencionamos, el Mapa se encuentra compuesto de muchos Casilleros, pero tambien mencionamos que existen distintos tipos de casilleros que actuan de forma distinta y permiten o no ser ocupadas por ciertas Unidades. Esto presenta un problema, dado que el casillero no sabe que lo ocupa y para los chequeos de si se puede ocupar por un edificio o unidad determinada no seria orientado al paradigma preguntar el tipo de lo que lo va a ocupar o el tipo de casillero.

5.2.2. TipoCasillero

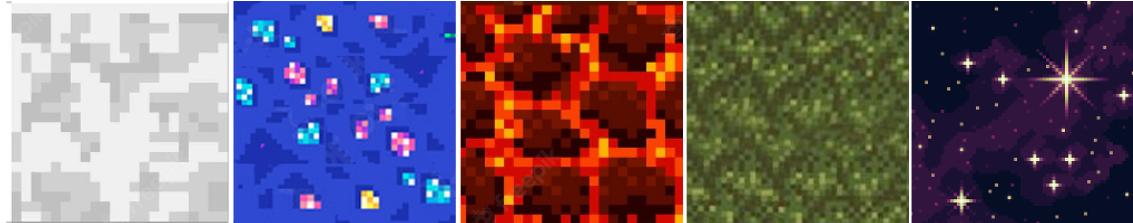


Figura 7: Imágenes utilizadas para TipoCasillero: Vacío, NodoMineral, NodoGas(Volcanes), Moho y CasilleroEspacial.

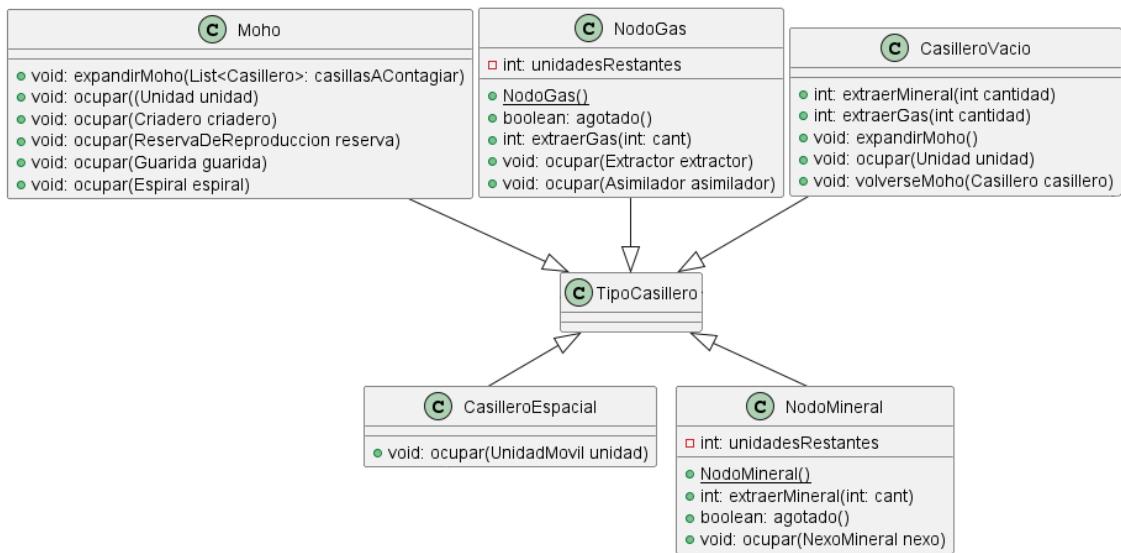


Figura 8: Diagrama de TipoCasillero

Para solucionar este problema se decidió implementar una clase `TipoCasillero`, para cada tipo de casillero distinto mencionado en la consigna y agregando un casillero vacío como tipo inicial. Inicialmente se iba a aplicar el Patrón Double Dispatch para `Ocupar`, la función que utilizan las unidades para avisar que quieren ponerse sobre la casilla, pero debido a la forma que se implementa la creación de Edificio, solo se utilizó una parte del patrón que es la de múltiples funciones que reciban parámetros distintos y que cada tipo de casillero haga un Overwrite en base a lo pedido. De esta forma, todas las unidades usan la misma función `Ocupar(this)` pero casillero hace la distinción y puede responder en base a lo esperado, ya sea con una excepción en caso de que no sea posible que esa unidad ocupe la casilla u ocupandola.

Finalmente, una forma similar es utilizada para expandir moho, puesto que todos reciben el mismo mensaje, pero Casillero Vacío es el único que puede transformarse en Moho cuando este se propaga. Una explicación más detallada del proceso se muestra en los diagramas de secuencia incluidos al final del informe.

5.3. Implementacion de Jugador

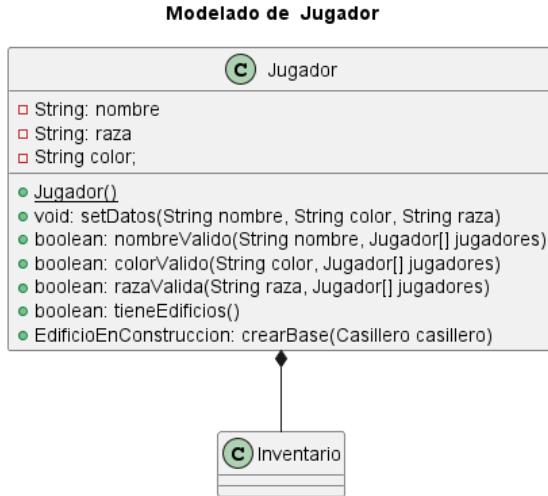


Figura 9: Diagrama de Jugador

Jugador tiene el inventario, los datos de cada usuario que juega y chequea la validez de los mismos. En el inventario se encuentran todas las unidades, los suministros y materiales de construcción de los que dispone. Para saber quién gana el juego, juego le pregunta al jugador si cumple las condiciones y este a su vez le delega a inventario, ya que ganar el juego es que el contrincante no tenga mas Edificios.



A su vez cada jugador en base a la raza que sea, inicia con un Criadero o un Pilón.

5.4. Implementación de Inventario

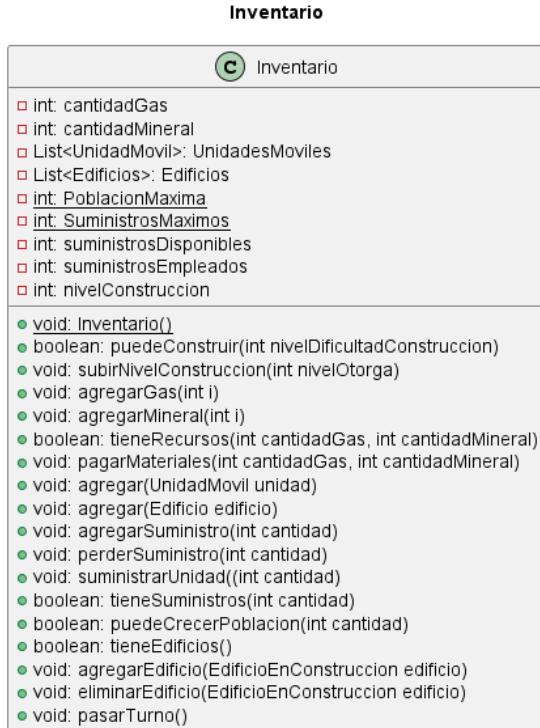


Figura 10: Diagrama de Inventario

Inventario es una de las clases que más presentes se encuentran a lo largo del modelo. Es donde se almacenan los recursos recolectados, el que paga los costos de las Unidades tanto en materiales como en suministros y es donde se tiene constancia de todas las unidades que tiene un jugador. A su vez es donde se detectan excepciones tales como no poder pagar o tener poblacion máxima (por lo que se distinguen suministros disponibles para gastar de los suministros empleados).

Uno de los problemas que afrontamos, fue que que se necesitaban tener ciertas construcciones para poder construir otras, lo cual inicialmente nos llevaba a un problema de tener que preguntar si se tenia un edificio de determinada clase, lo cual no sería acorde al paradigma que buscamos emplear. Esto fue solucionado a partir de darle a las construcciones un nivel requerido para construirse y un nivel que otorgan una vez que las construís por primera vez. Dado que, a la hora de construir un edificio ya de por sí se debe tener el inventario para pagar la construcción, decidimos almacenar los datos de Nivel de construcción del jugador en el mismo, para poder realizar los chequeos de correlatividad antes de construir.

5.5. Implementacion de Unidades

Otro de los puntos principales del juego son las unidades: decidimos dividirlas en edificios y unidades moviles, y dentro de las mismas dividirlas en Zerg y Protoss. Tratamos de aislar las responsabilidades en sus propias clases e interfaces, puesto que es uno de los elementos que tiene que cumplir con la mayor cantidad de requisitos y que cuenta con la mayor cantidad de características.

5.5.1. Unidades Moviles

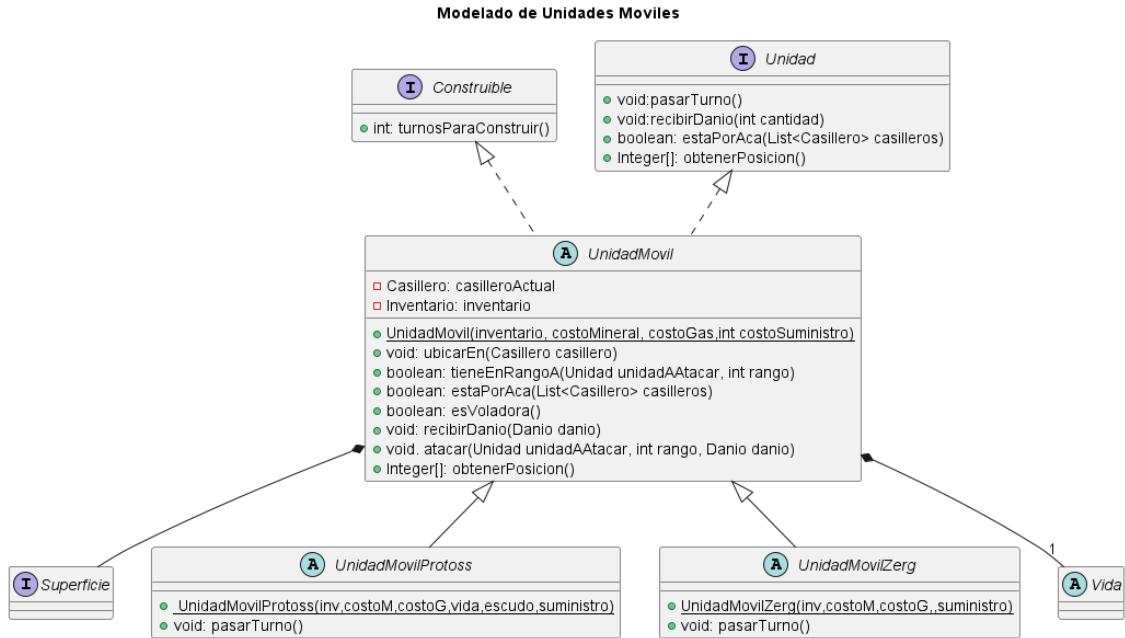


Figura 11: Diagrama general de Unidades Moviles

Cada unidad implementa dos interfaces: Construible y Unidad. De esta forma buscamos tratar de dividir lo mas posible los comportamientos y agrupar aquellos similares. La clase abstracta Unidad Movil tiene una superficie y una vida, que serán explicadas en detalle en las proximas secciones. Unidad Movil Protoss y Unidad Movil Zerg heredan de UnidadMovil diferenciandose en su comportamiento al pasar de turno y se destaca que en la creacion uno tiene vida y escudo y el otro solamente cuenta con vida.

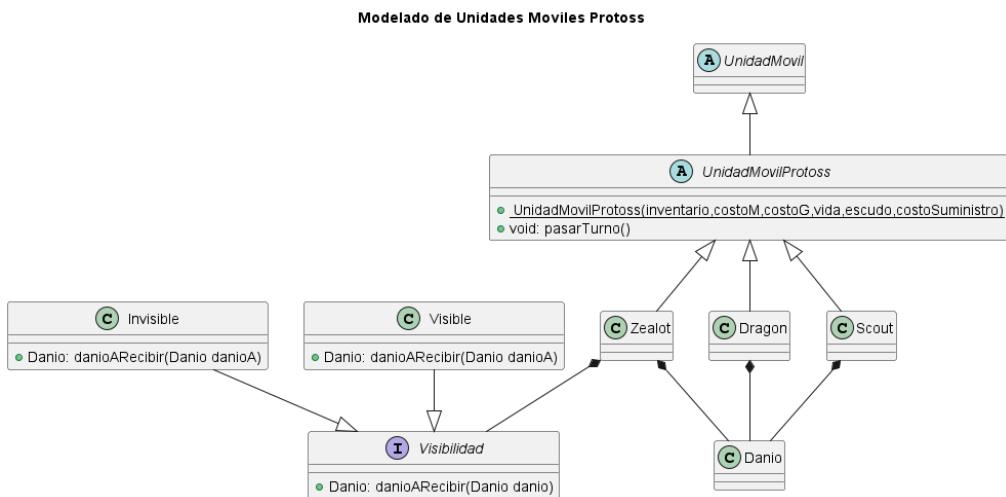


Figura 12: Diagrama de Unidades Moviles Protoss. El diagrama se encuentra simplificado.

Como se puede ver en la figura, en el caso de las Unidades Moviles Protoss cada uno tiene un

daño distinto y el Zealot tiene la distinción de poder ser invisible, para lo que implementamos la interfaz Visibilidad, a partir de la cual, recibe de formas distintas el daño.

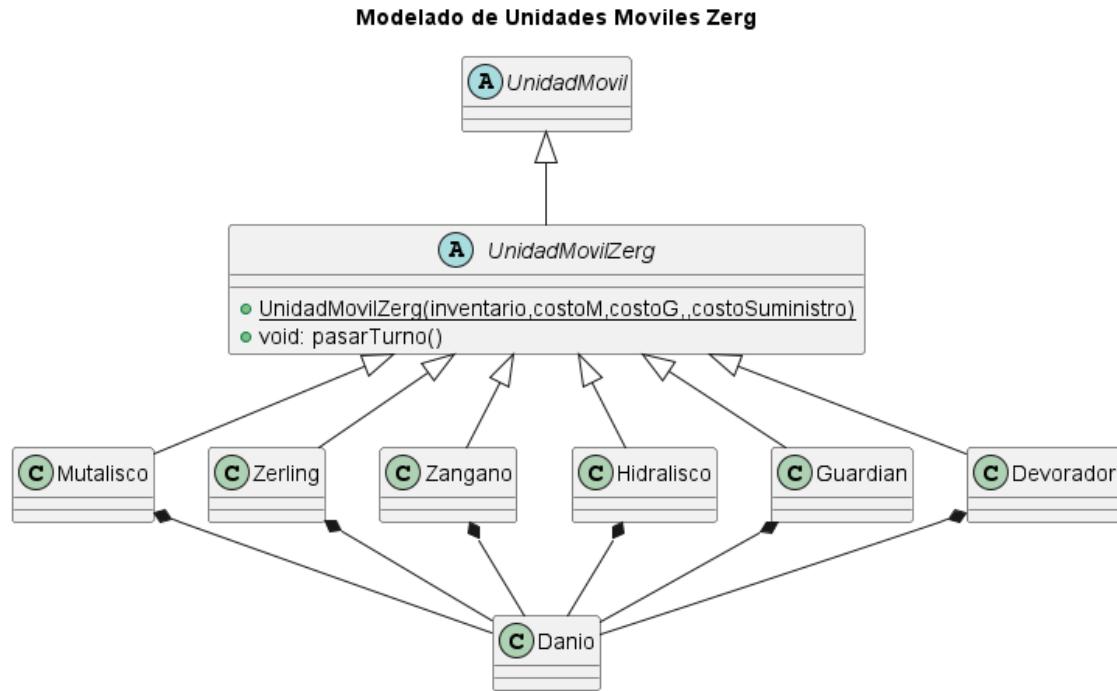


Figura 13: Diagrama de Unidades Móviles Zerg. El diagrama se encuentra simplificado para su comprensión.

5.5.2. Edificios

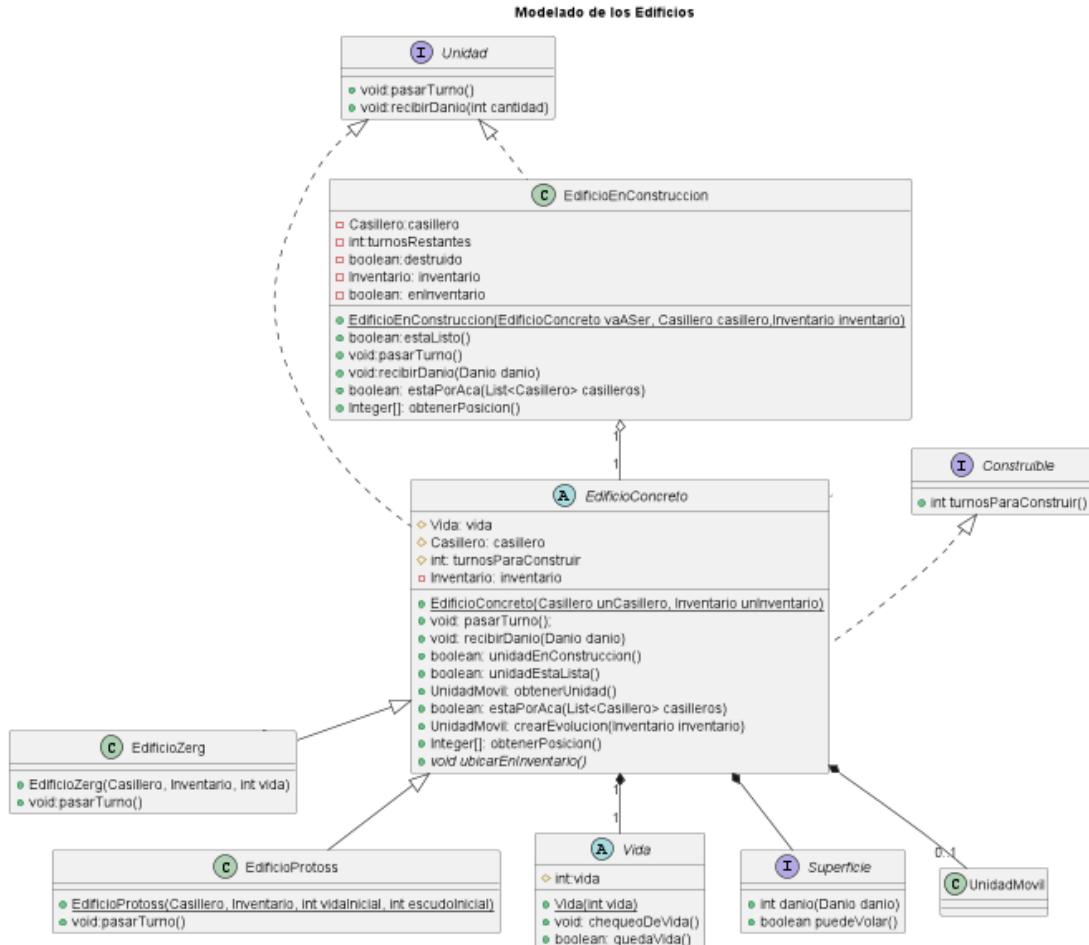


Figura 14: Diagrama de Unidades y Edificios

Los EdificiosEnConstrucción implementan la interfaz Unidad y tienen en agregación un EdificioConcreto: cuando se construye un edificio, inicialmente se crea un edificio en construcción que contiene un edificio concreto que será un edificio Zerg o un edificio Protoss, en definitiva, el edificio final. De esta forma una vez que pasan los turnos necesarios para que se termine de construir, si se le mandan mensajes al edificio, estos se delegarán al edificio concreto. Los edificios concretos, al igual que las unidades móviles, implementan Unidad y Construible. Por otro lado, también tienen una Vida y una Superficie, las cuales serán explicadas en detalle en las siguientes secciones y también, en algunos casos, pueden contener una UnidadMovil.

A continuación mostraremos los diagramas de clase de los distintos Edificios Protoss y Edificios Zerg.

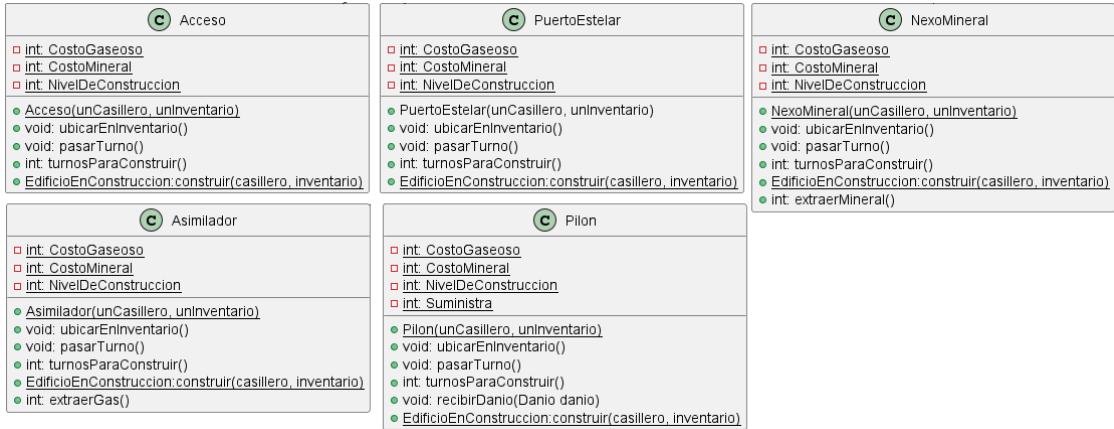


Figura 15: Edificios Protoss. Estos heredan de la clase EdificioProtoss

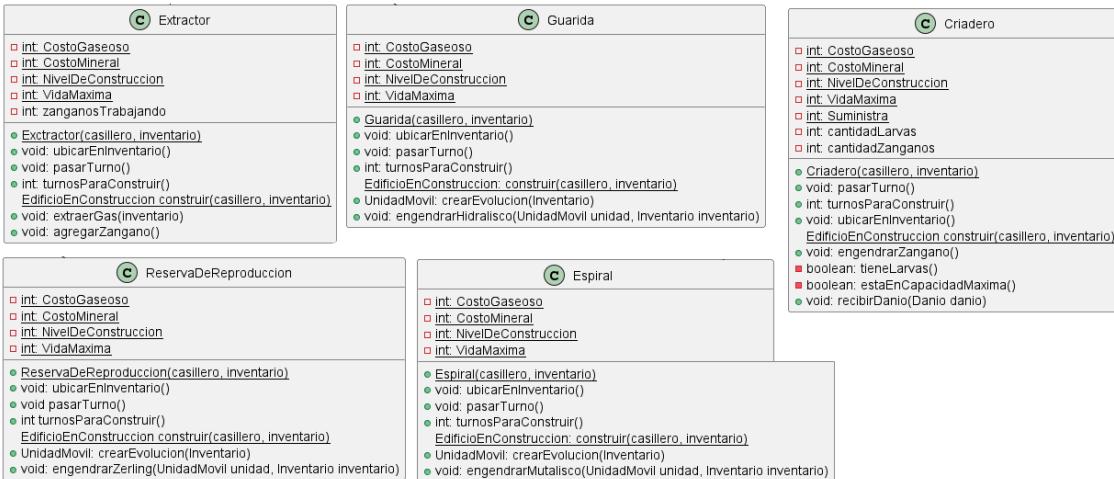


Figura 16: Edificios Zerg. Estos heredan de la clase EdificioZerg

5.5.3. Vida, Escudo, Danio y Superficie

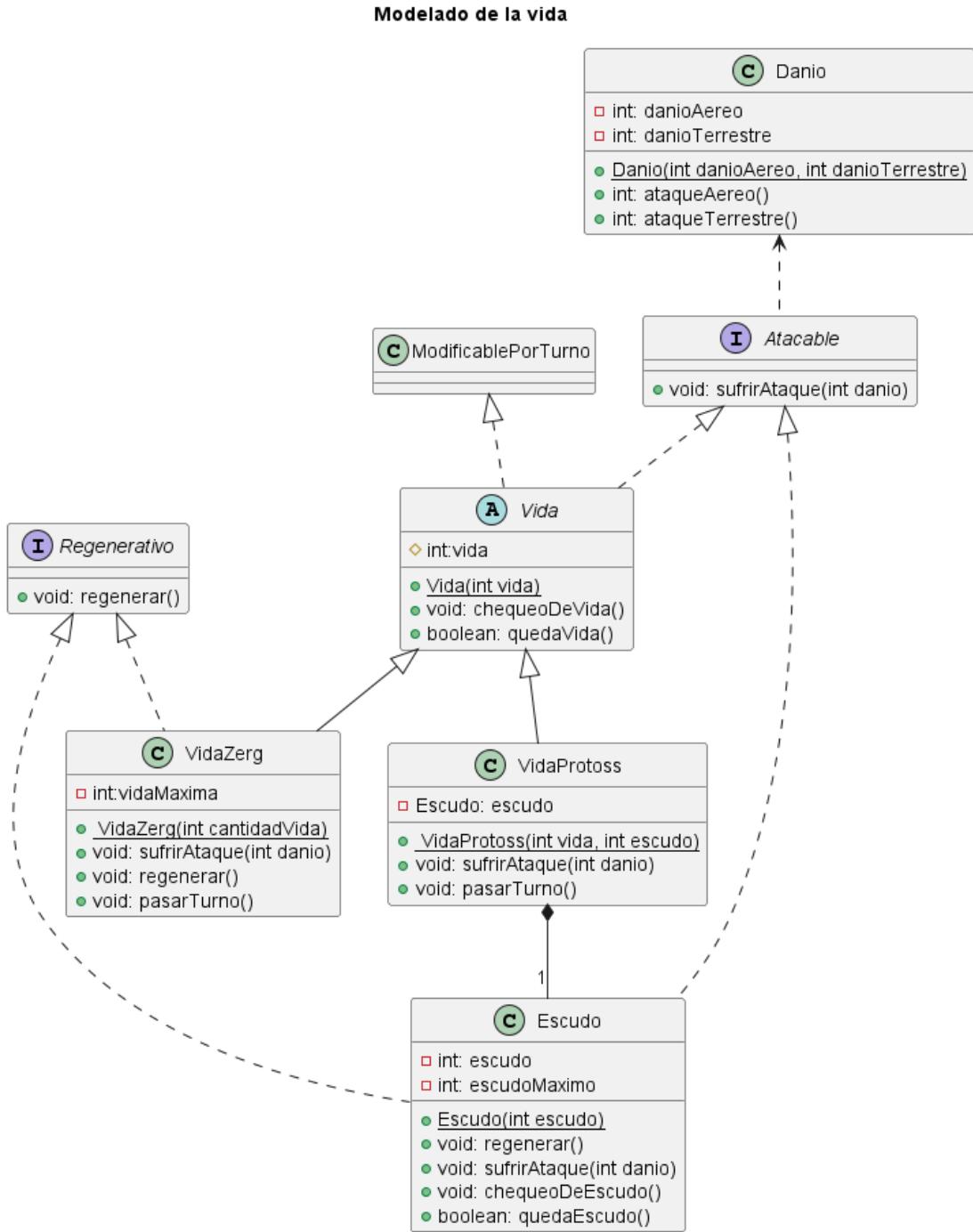


Figura 17: Diagrama de Vida.

Como ya mencionamos anteriormente las Unidades tienen Vida y pueden o no tener escudo. Sumado a eso solo la vida Zerg y los escudos se regeneran, por lo que se implementó la interfaz Regenerativo. Tanto la vida como el escudo es atacable y en vez de recibir un valor, recibe un

daño. La necesidad de implementar daño surgió de que se tienen dos tipos de daños, el daño aereo y el daño terrestre.

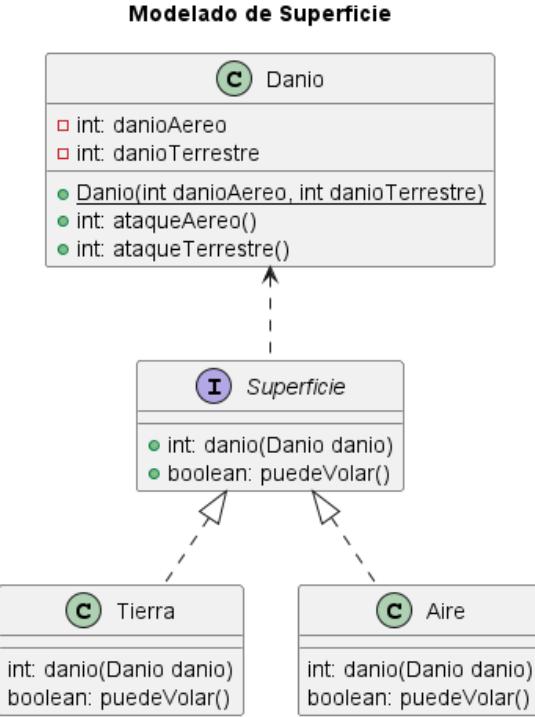


Figura 18: Diagrama de Superficie.

Así como se tiene daño, cada unidad del juego tiene que instanciar una Superficie. Esta se instancia como Tierra o Aire, donde cuentan con un método para recibir daño. De esta forma sabemos cuándo daño corresponde a esa superficie para el daño recibido por parámetro.

6. Excepciones

AtaqueFueraDeRango Excepción lanzada cuando se busca atacar a una unidad que no se encuentra en el rango de ataque del atacante.

CorrelativasInsuficientes Excepción lanzada cuando no se tienen las Unidades que se requieren como pre-requisito para construir una determinada unidad nueva.

EdificioOcupado Excepción lanzada cuando se busca engendrar una unidad en un edificio que ya se encuentra ocupado engendrando otra cosa.

EstaDestruido Excepción lanzada cuando una Unidad no tiene mas vida o escudo.

ExtractorError Excepción lanzada cuando se pide extraer recursos a un Extractor que no tiene zanganos que puedan hacerlo.

ParametrosInvalidos Excepción lanzada cuando los parametros ingresados para registrar un jugador son invalidos.

PoblacionMaximaAlcanzada Excepción lanzada cuando se quiere construir una Unidad que consume suministros cuando la cantidad maxima de suministros empleados es alcanzada, es decir, la población maxima.

RecursosInsuficientes Excepción lanzada cuando se intenta pagar algo para lo que no se cuentan con los recursos suficientes.

SuministrosInsuficientes Excepción lanzada cuando se intenta pagar algo para lo que no se cuentan con los suministros suficientes.

UbicacionInvalida Excepción lanzada cuando se quiere ocupar una casilla que no cumple con los requisitos o ya esta ocupada.

UnidadInvisible Excepción lanzada cuando se intenta dañar una unidad invisible, puesto que estas no pueden recibir daño.

UnidadOcupada Excepción lanzada cuando la unidad ya se encuentra ocupada.

UnidadYaEvolucionada Excepción lanzada cuando la unidad no puede evolucionar.

YaNoQuedanLarvas Excepción lanzada cuando un criadero no cuenta con mas larvas.

7. Diagramas de secuencia

7.1. Expandir Moho al pasar turno

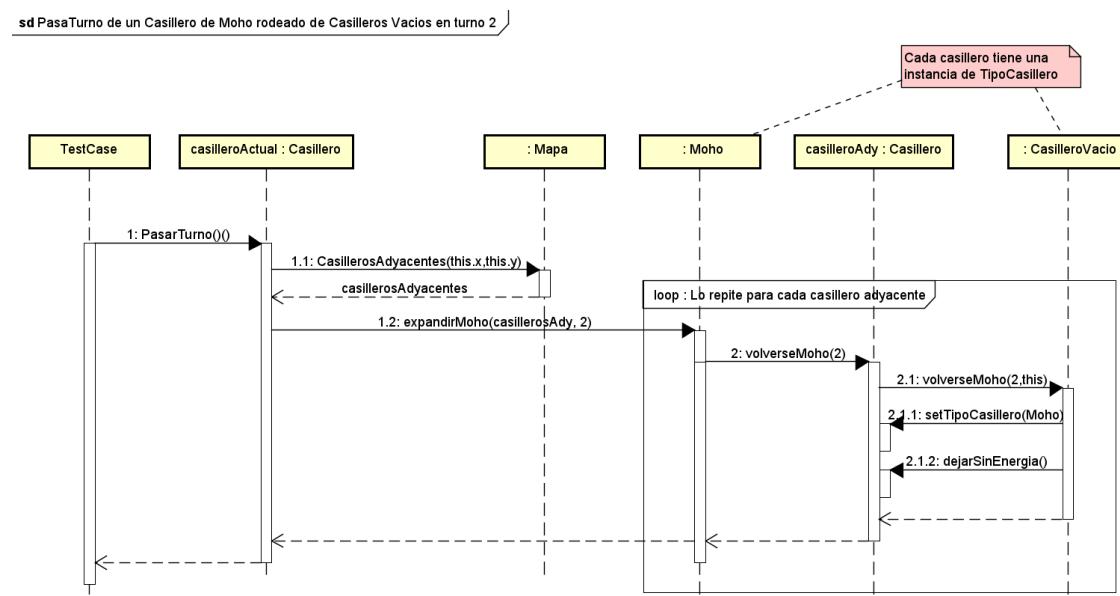


Figura 19: Diagrama de una casilla pasando de turno.

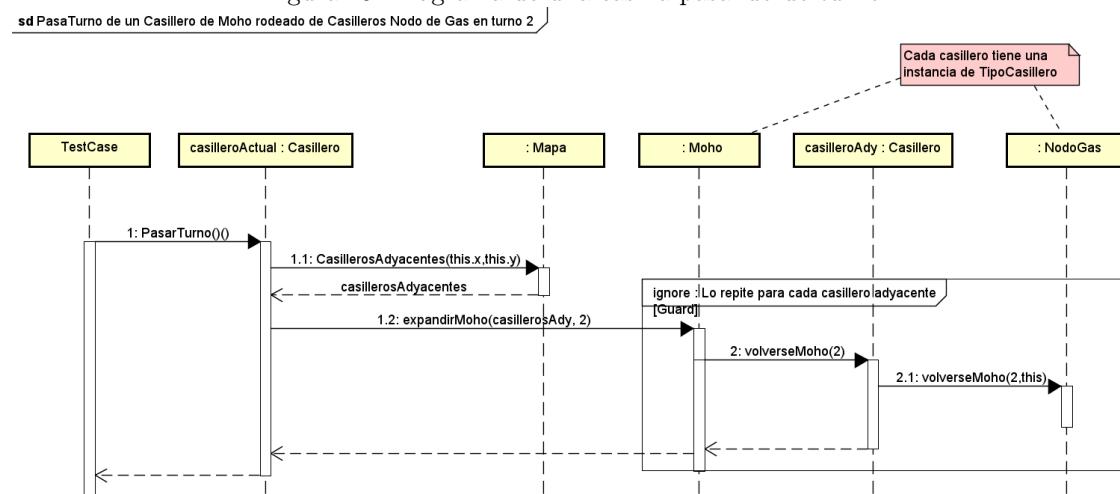


Figura 20: Diagrama de una casilla pasando de turno, alternativa cuando lo que rodea no es una casilla que se pueda volver moho.

7.2. Creacion de un Juego Nuevo, registro un jugador y creo un mapa listo para usarse

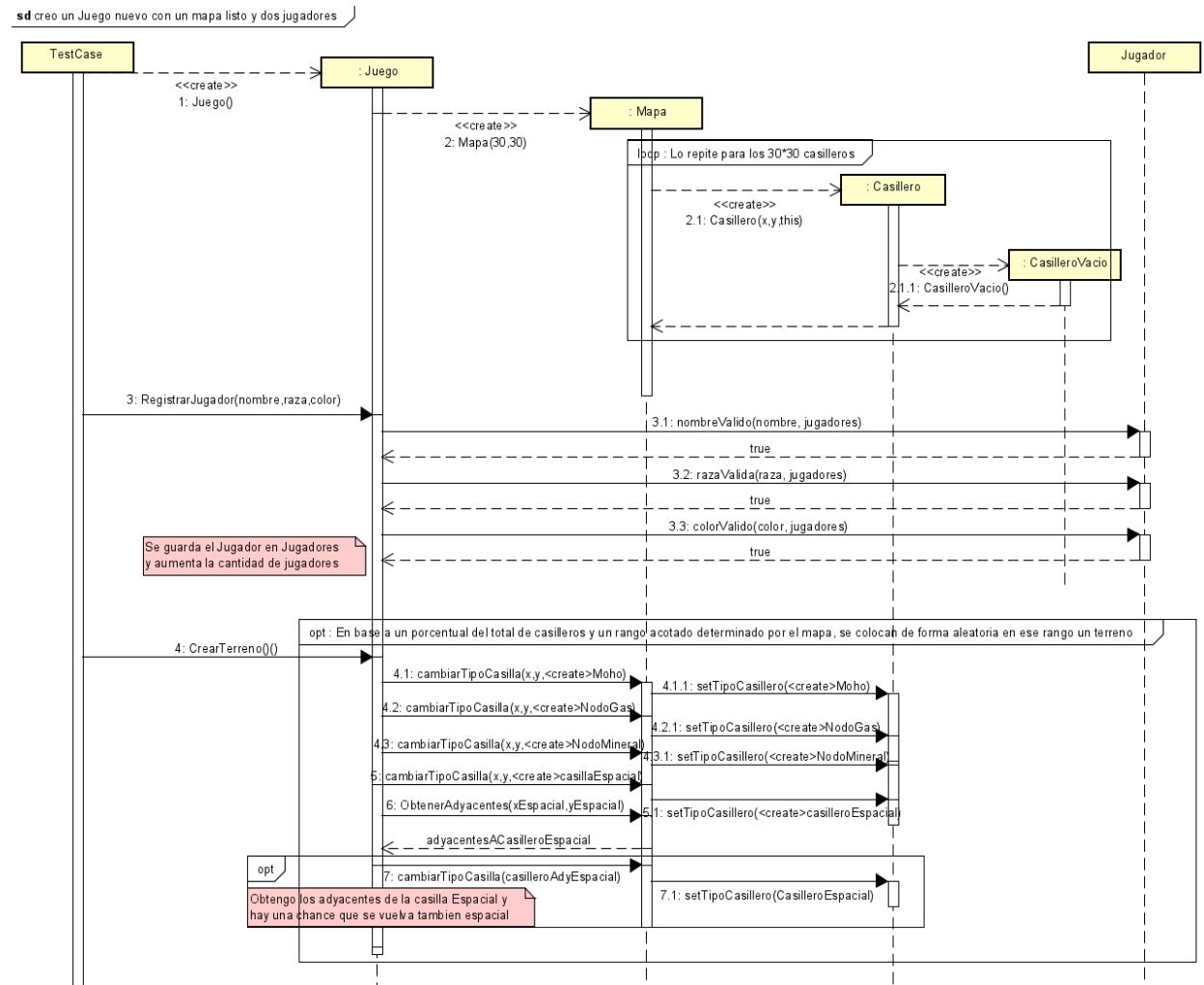


Figura 21: Diagrama de la creacion de un Juego, registrando jugador y preparando un Mapa.

7.3. Una Unidad Movil ataca un edificio y lo rompe

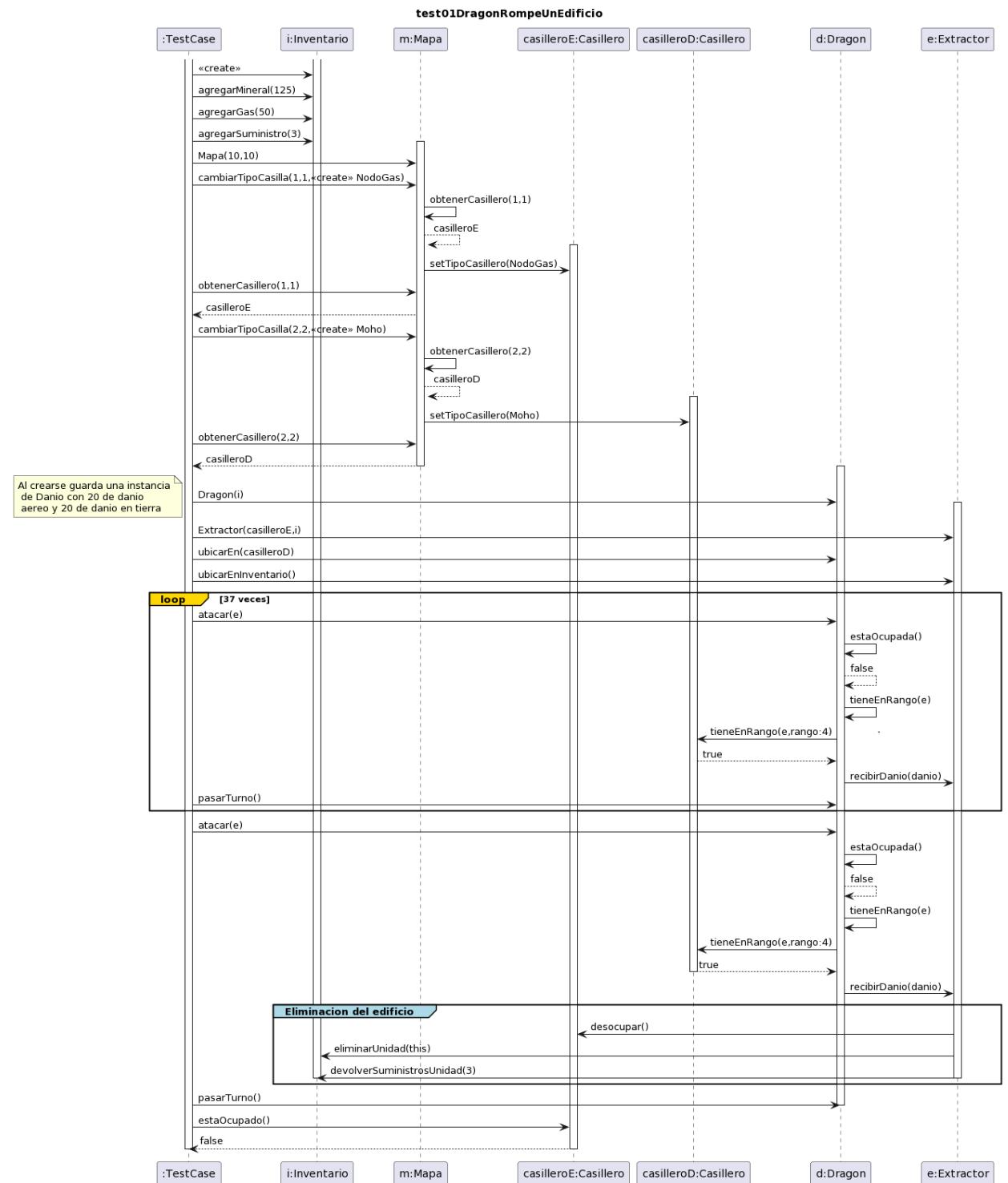


Figura 22: Diagrama de un Dragon atacando y rompiendo un extractor.

7.4. Creo un Edificio Zerg sobre un Casillero energizado

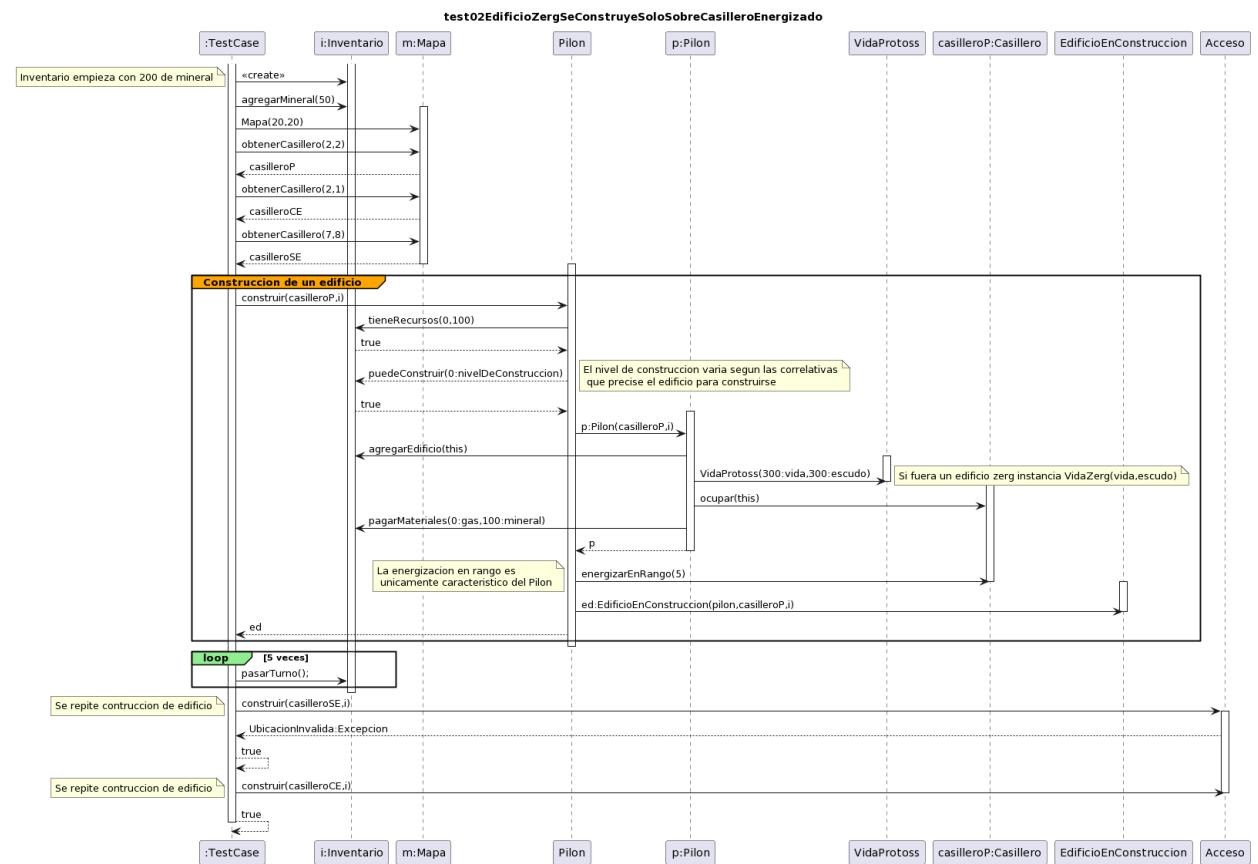


Figura 23: Diagrama de un Edificio Zerg.

7.5. Dos unidades pelean hasta que una muere

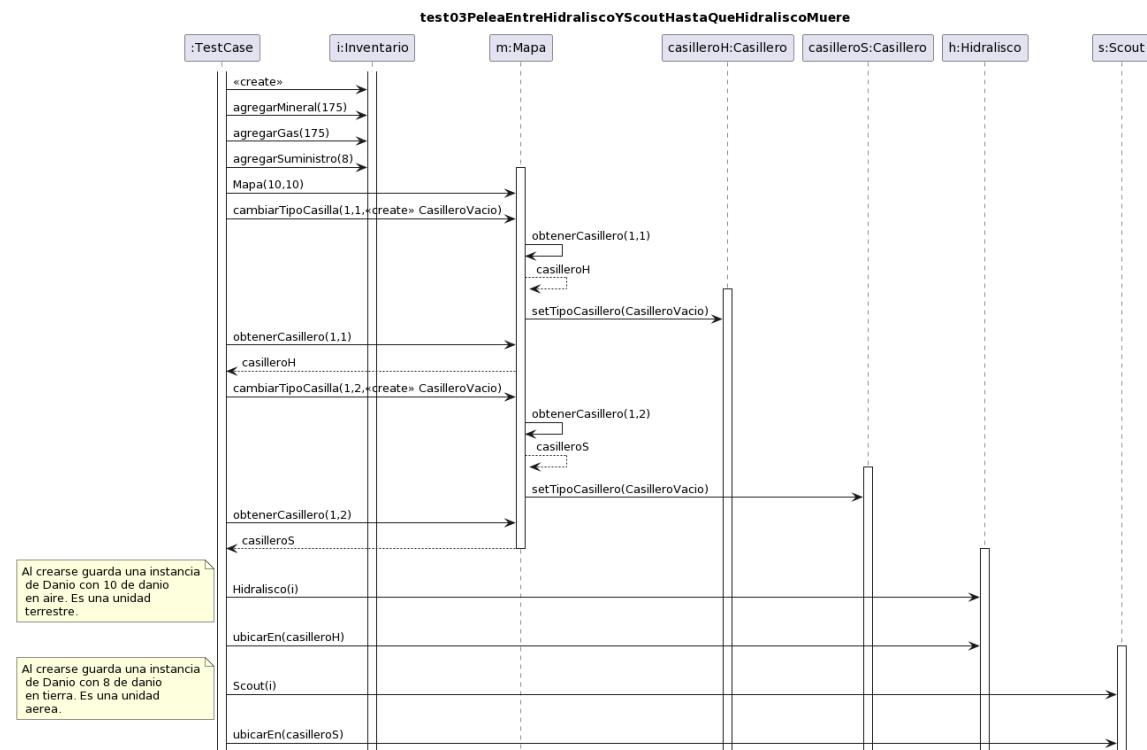


Figura 24: Primera parte de pelea entre unidades, instanciar clases.

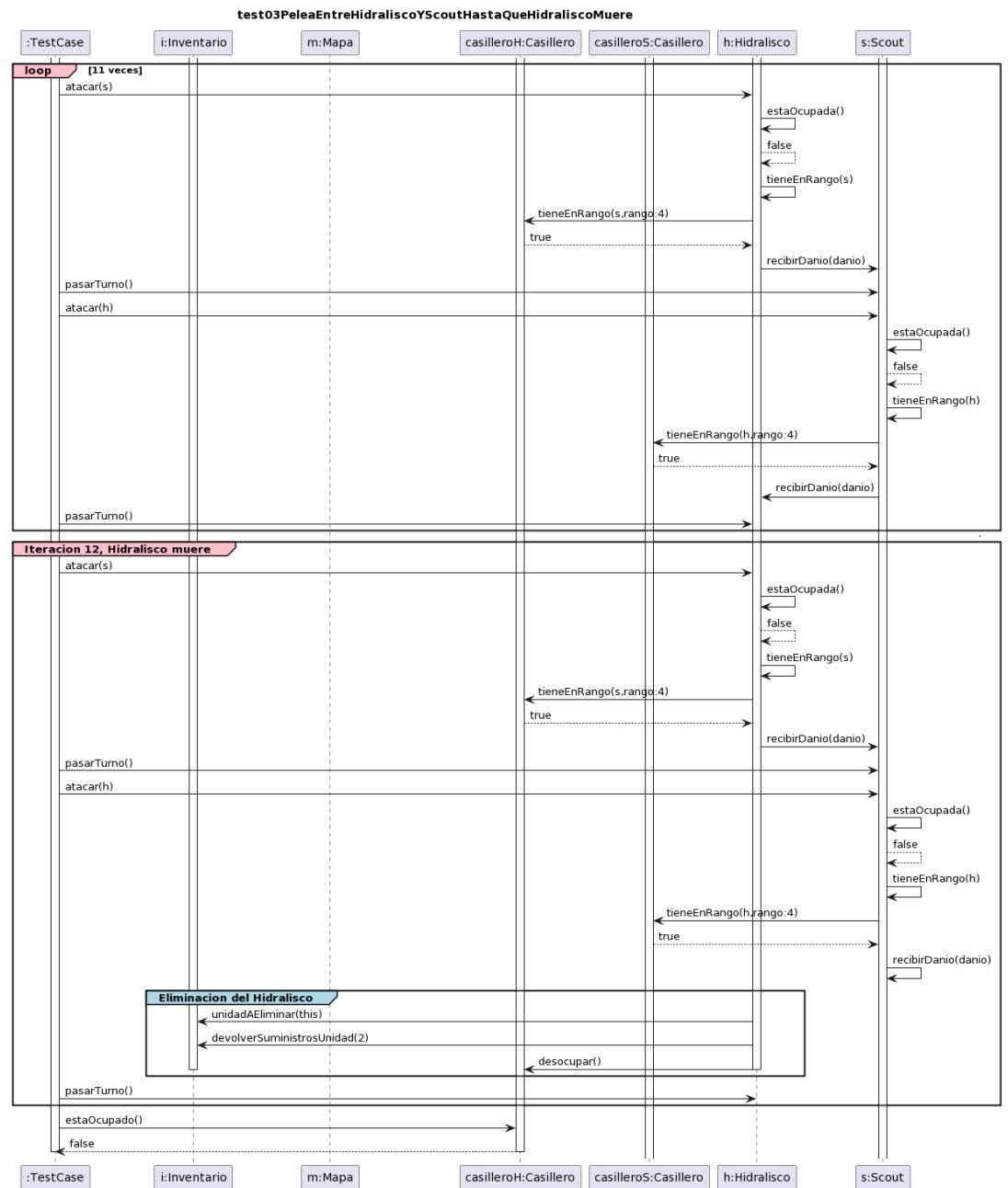


Figura 25: Diagrama de dos unidaes peleando hasta que una muere.