[75.07 / 95.02]
Algoritmos y programación III
Trabajo práctico 2: AlgoStar

(trabajo grupal)

Estudiantes:

Nombre	Padrón	Mail
Leonardo Duchen	104855	lduchen@fi.uba.ar
Brian Lahuta	98675	blahuta@fi.uba.ar
Cristian Leith	103494	cleith@fi.uba.ar
Alan Valdevenito	107585	avaldevenito@fi.uba.ar

Tutor: Santiago Valdéz

Nota Final:

Introducción y especificaciones

AlgoStar es un juego similar al famoso StarCraft pero no en tiempo real sino por turnos. El mismo es un juego de guerra de estrategia y se basa en la construcción y administración de un imperio.

Las razas

- Zerg: Una raza de insectoides alienígenas en busca de la perfección genética, obsesionada con la asimilación de otras razas.
- Protoss: Son una especie humanoide con tecnología avanzada y habilidades psiónicas, tratando de preservar su civilización.

Gestión de recursos

Durante el juego se deberá recolectar:

- Minerales: habrá nodos de minerales distribuidos por todo el mapa, cada nodo tiene 2000 unidades de minerales.
- Gas vespeno: habrá volcanes que expulsan dicho gas cerca de los nodos de minerales. Cada volcán tiene 5000 unidades de gas.

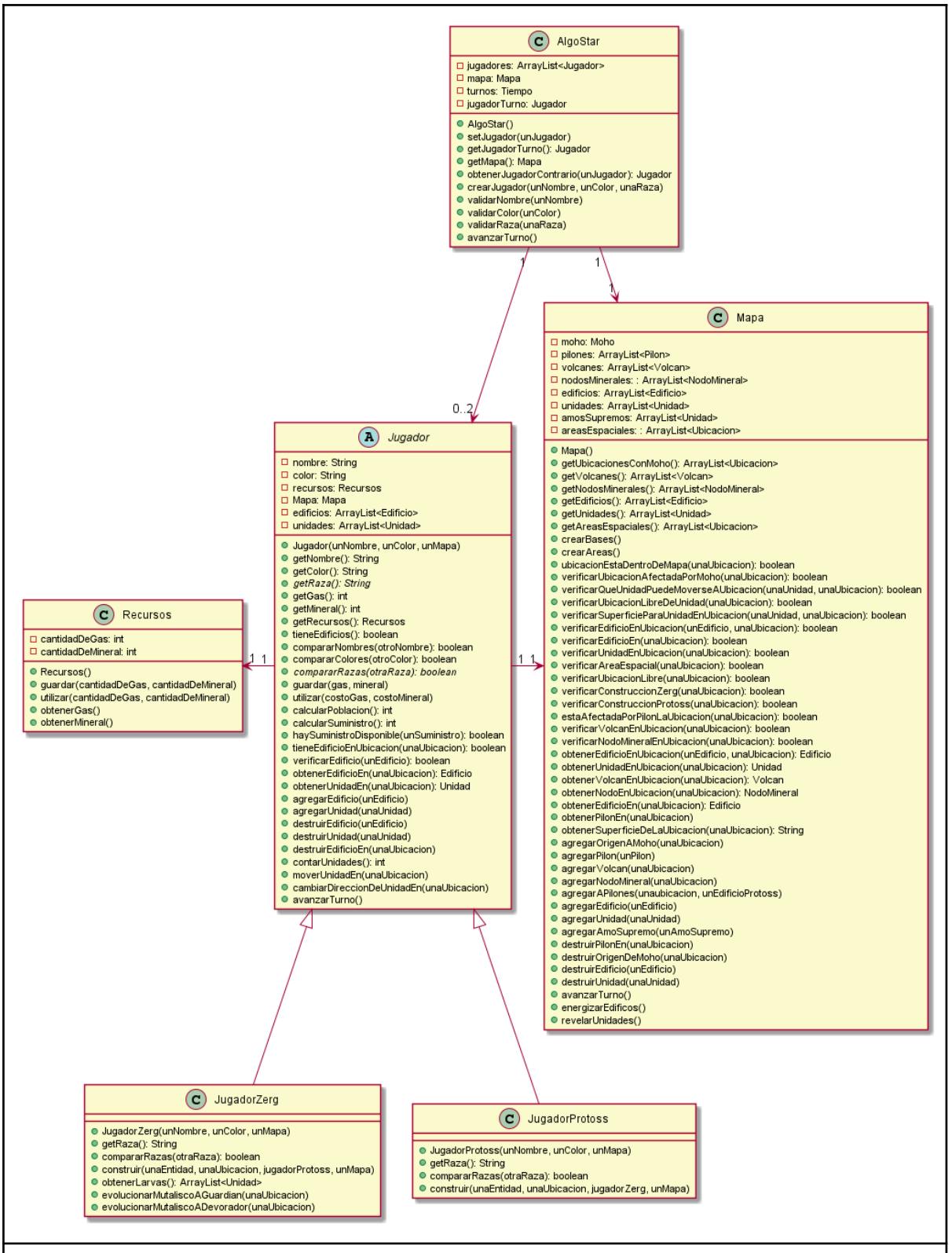
Para poder colectar el gas vespeno cada raza deberá construir una “refinería de gas” sobre el volcán dónde se origine el gas vespeno que desea extraer.

Supuestos

- Cristal y Nodo Mineral son lo mismo.
- Geiser y Volcán son lo mismo.
- Debería haber al menos dos bases al crear un mapa.
- Todos los edificios tienen superficie de tipo tierra.
- Todas las ubicaciones tienen por default superficie de tipo tierra.
- Nexo Mineral extrae 10 unidades de mineral por turno.
- Cada edificio Zerg regenera su vida un 5% por cada turno que pasa.
- Cada edificio Protoss regenera su escudo un 5% por cada turno que pasa.
- El jugador Zerg debe crear obligatoriamente primero un Criadero para poder construir otros edificios debido al moho, por esta razón es que se gasta sus 200 de mineral iniciales y no tiene más recursos. Tampoco tiene forma de conseguir más. Por esta razón en cada turno los jugadores aumentan sus recursos en +10.
- Aunque un edificio no se encuentre operativo podrá recibir daño y regenerarse.

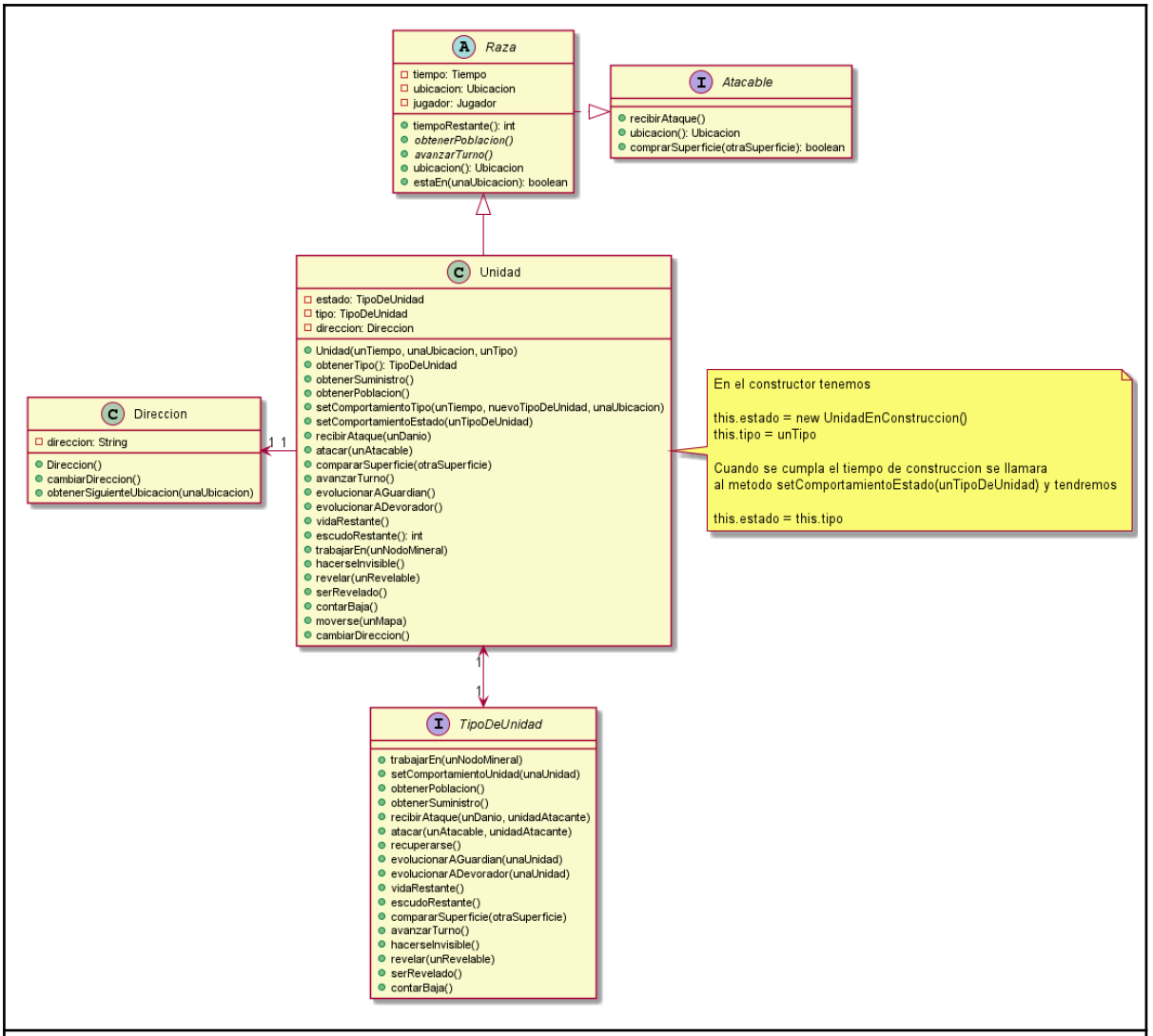
Diagramas de clases

AlgoStar



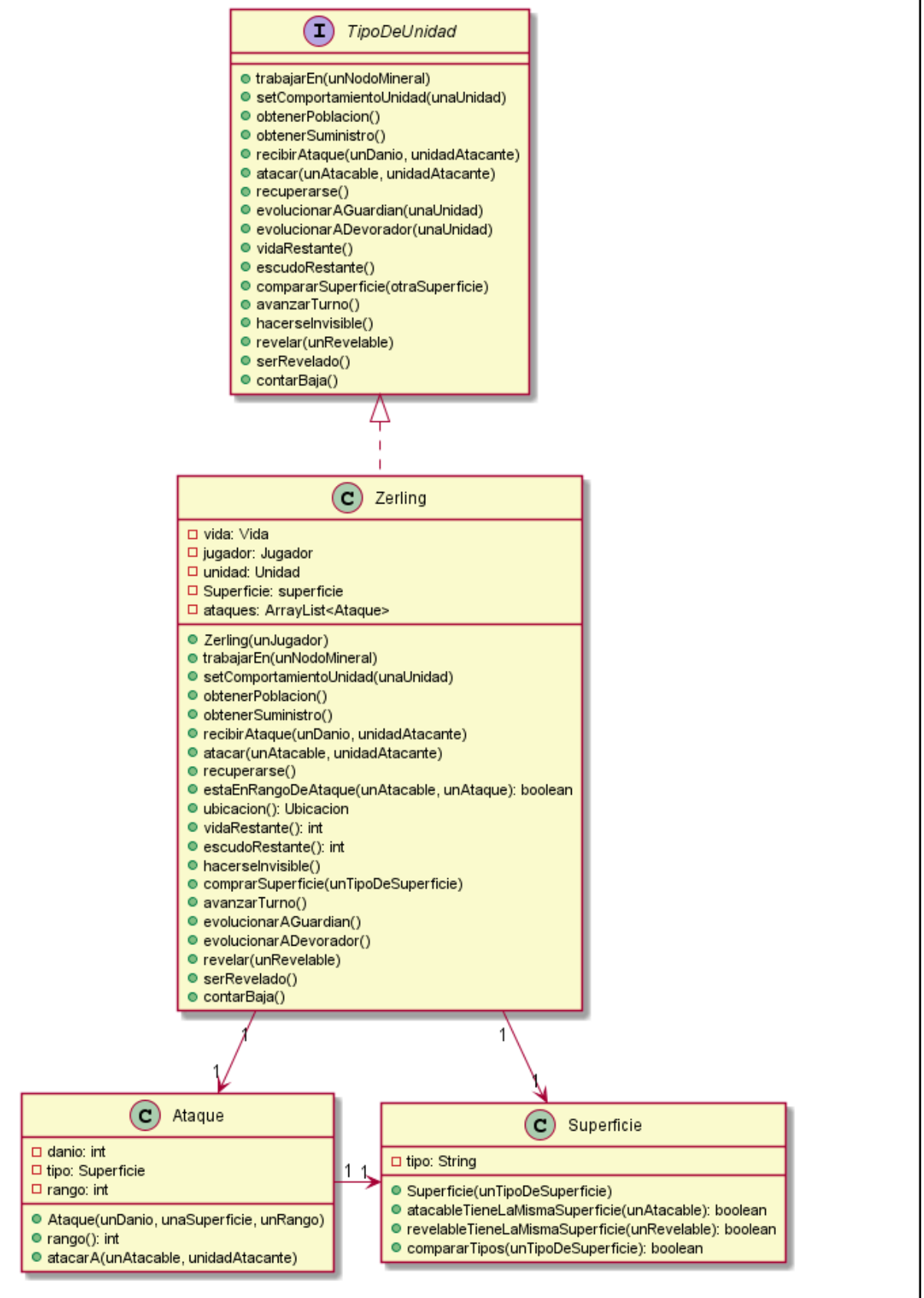
AlgoStar

Unidad



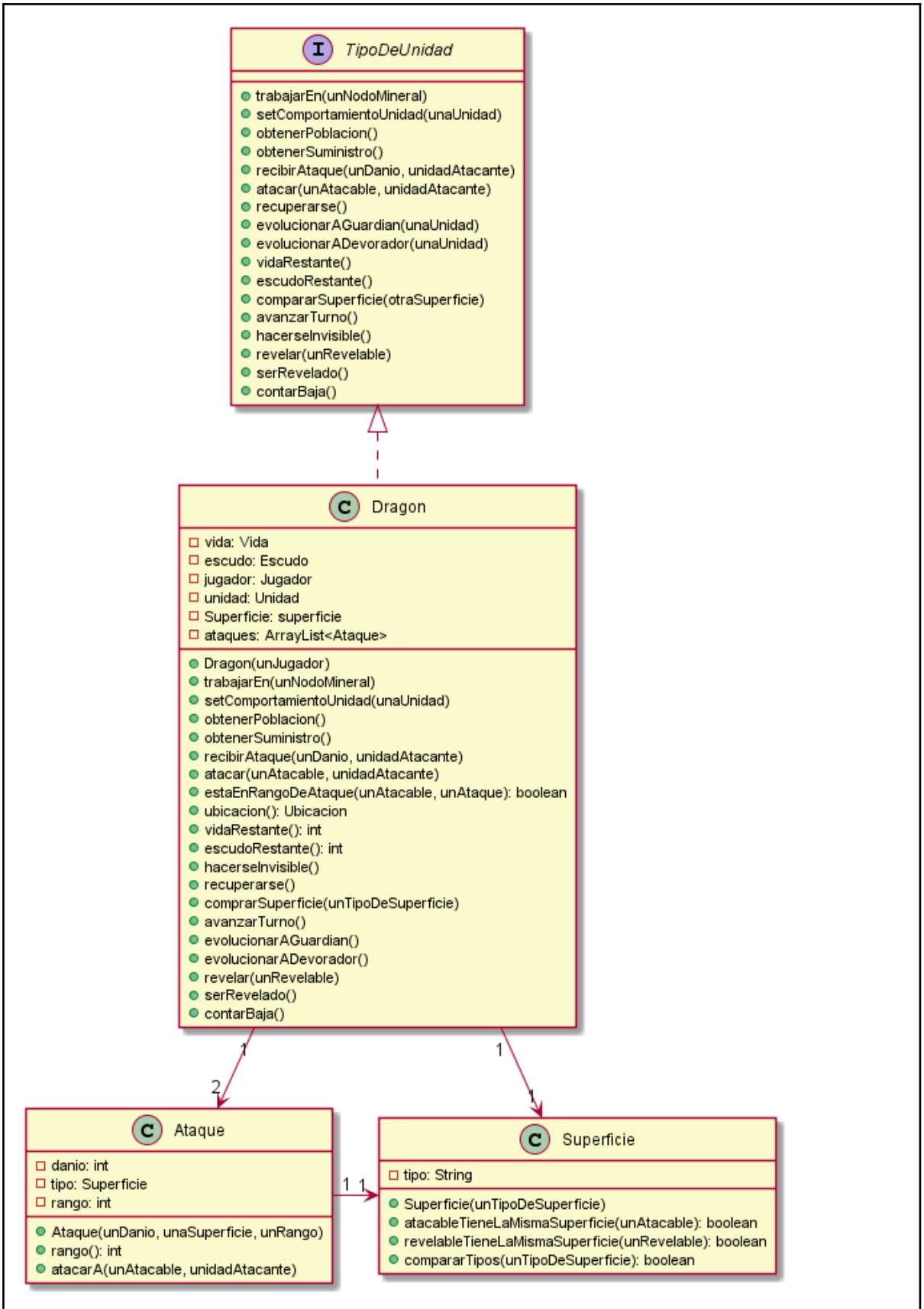
Unidad

Unidad Zerg



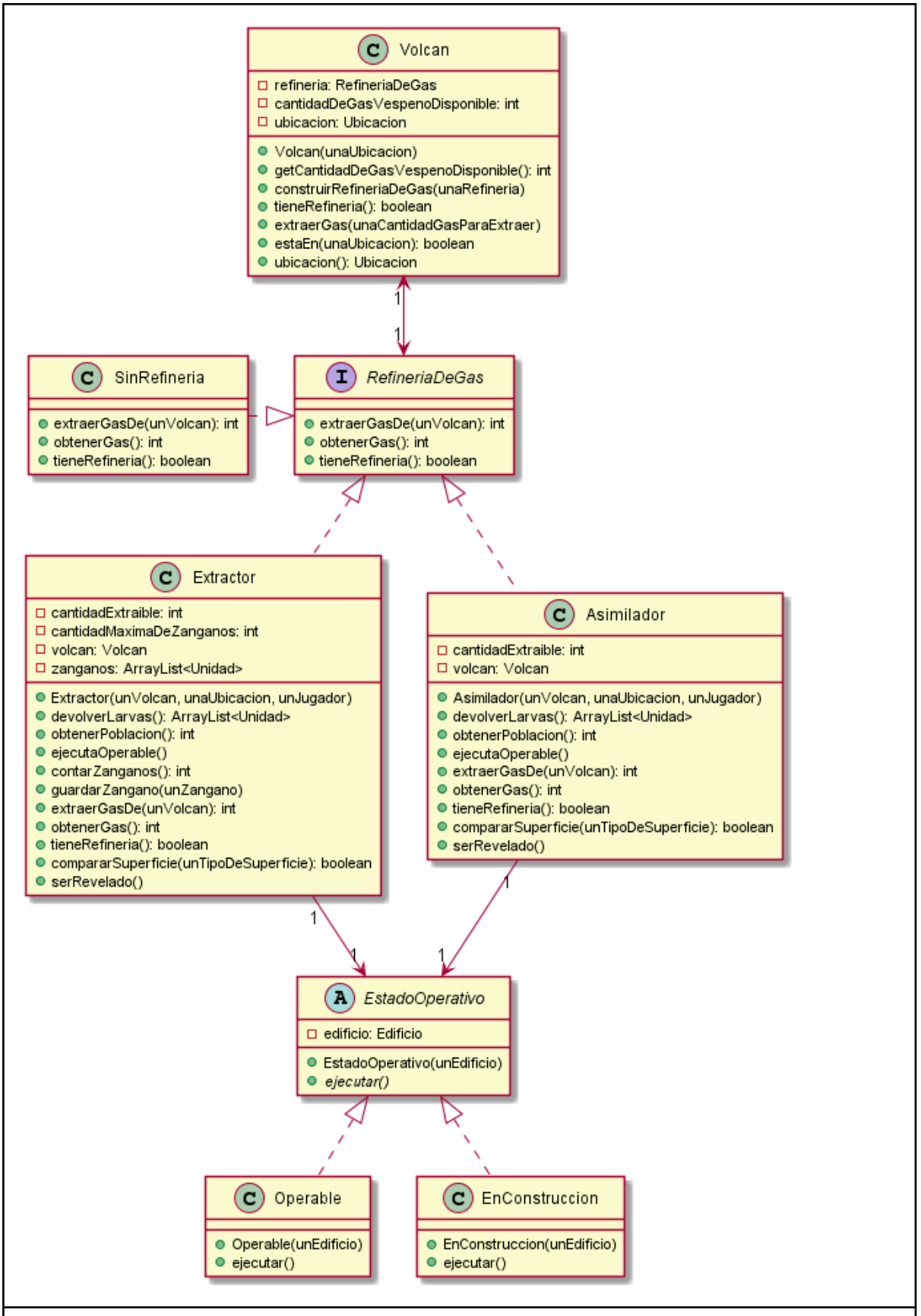
Unidad Zerg: un Zerling

Unidad Protoss



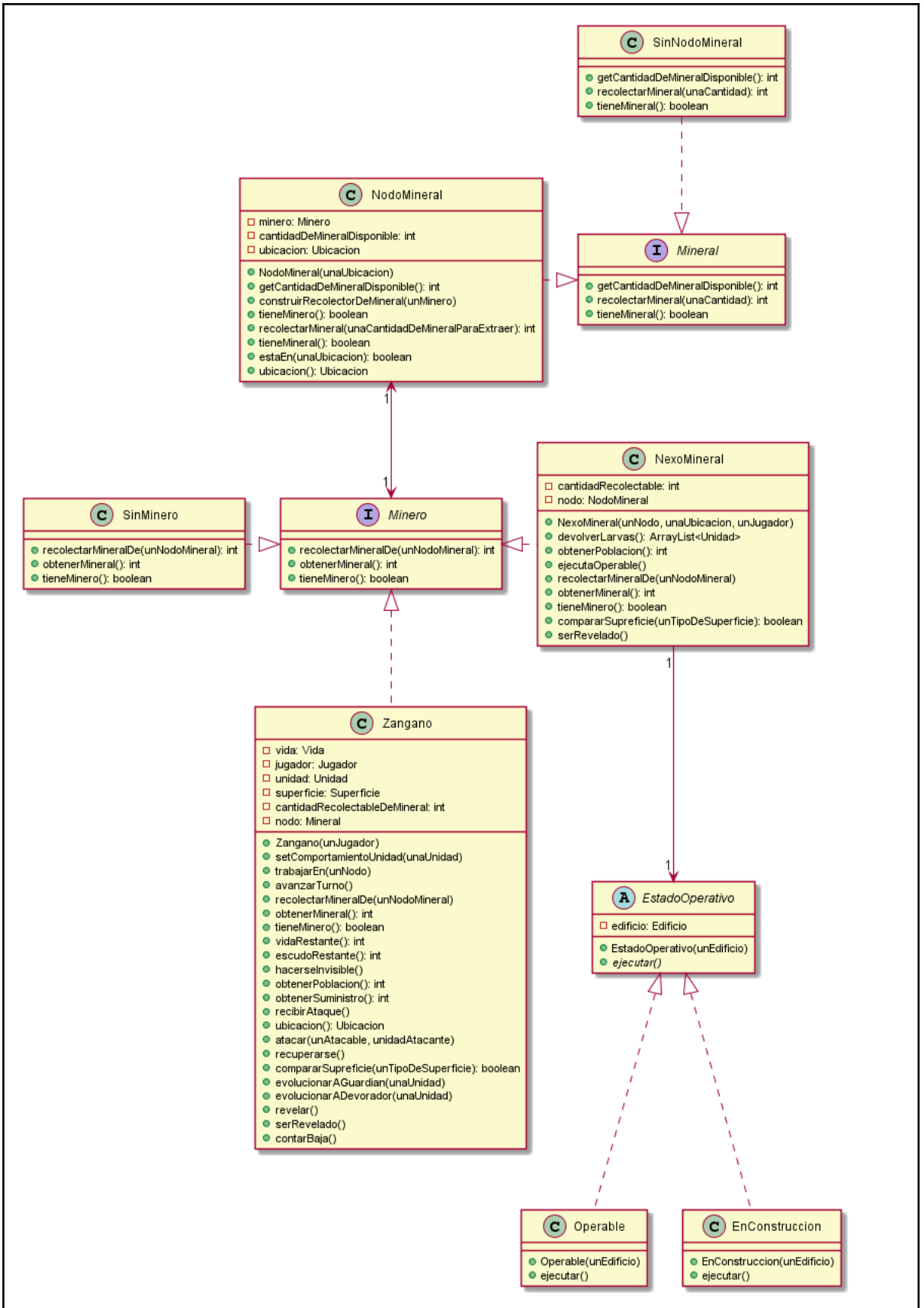
Unidad Protoss: un Dragón

Volcán



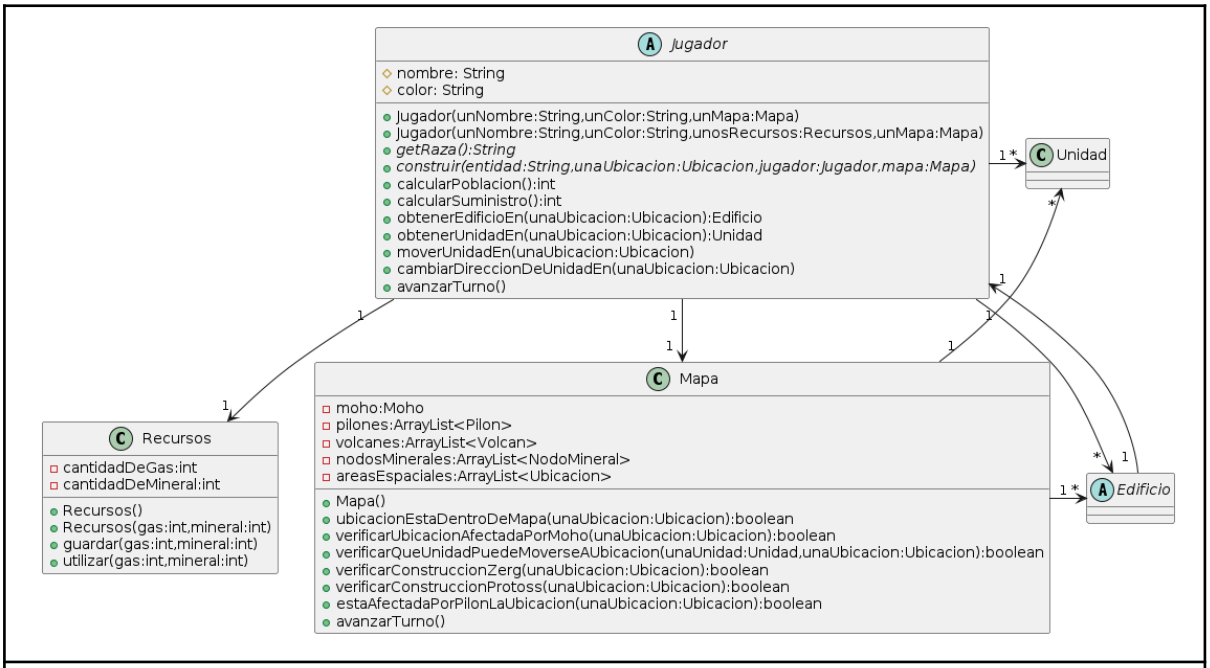
Volcán

Nodo mineral



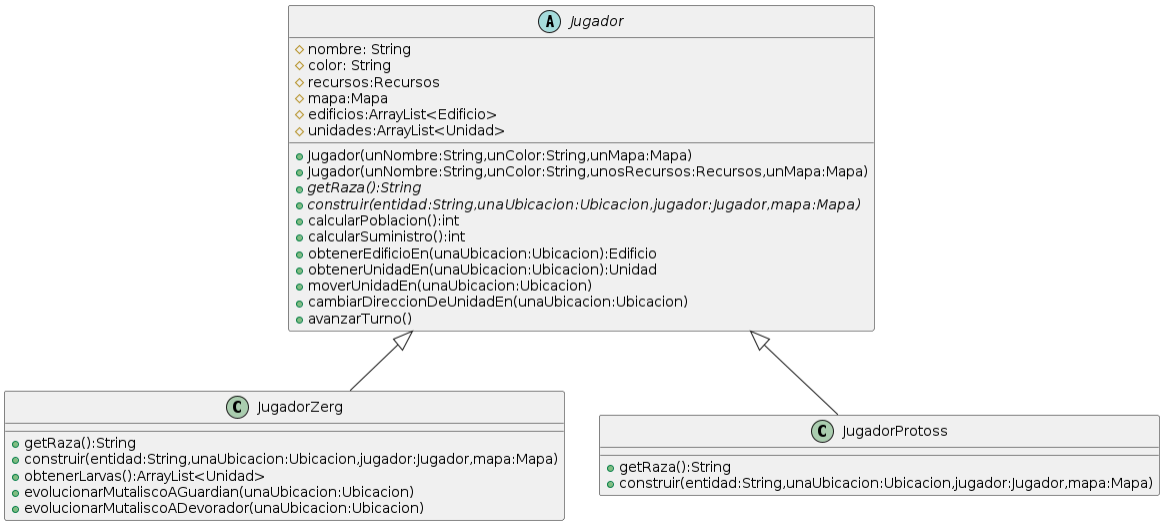
Nodo mineral

Jugador



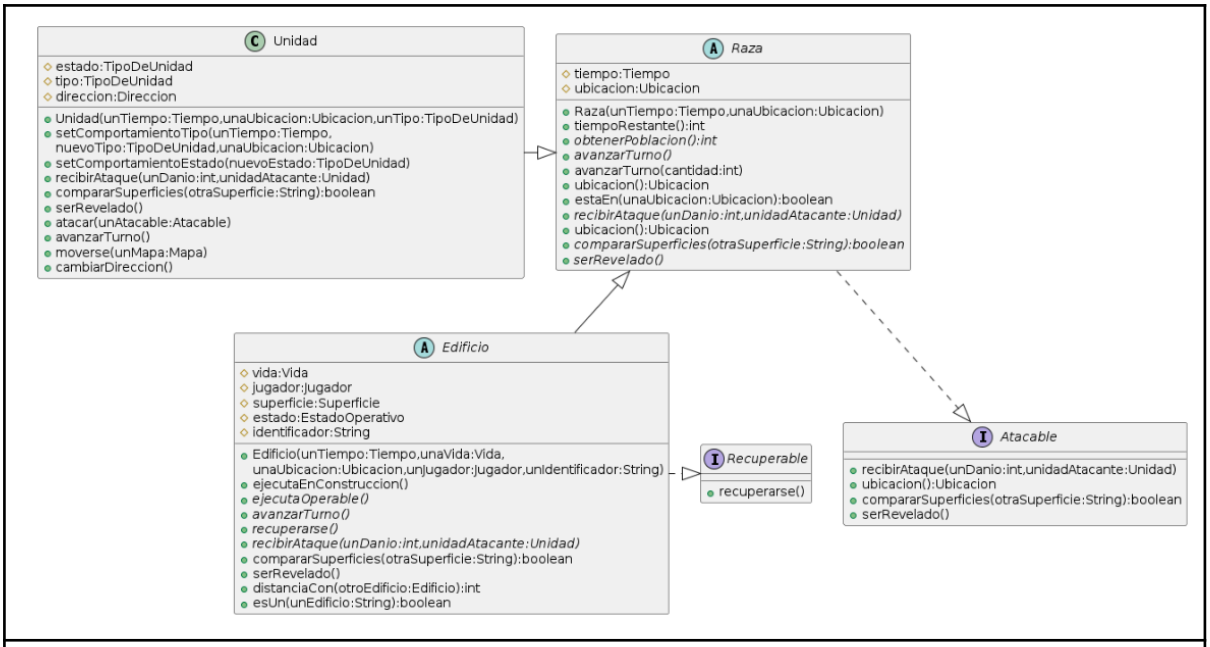
Relación entre Jugador y sus clases internas

Jugadores



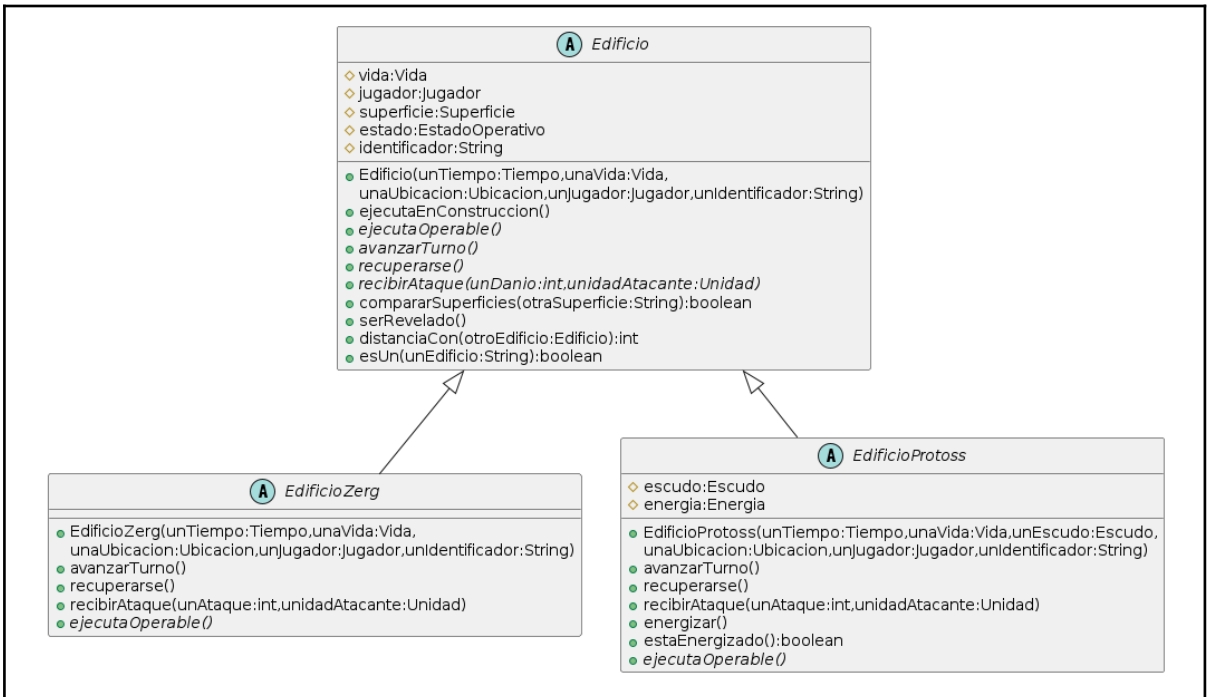
Relación entre Jugador y sus clases hijas

Raza



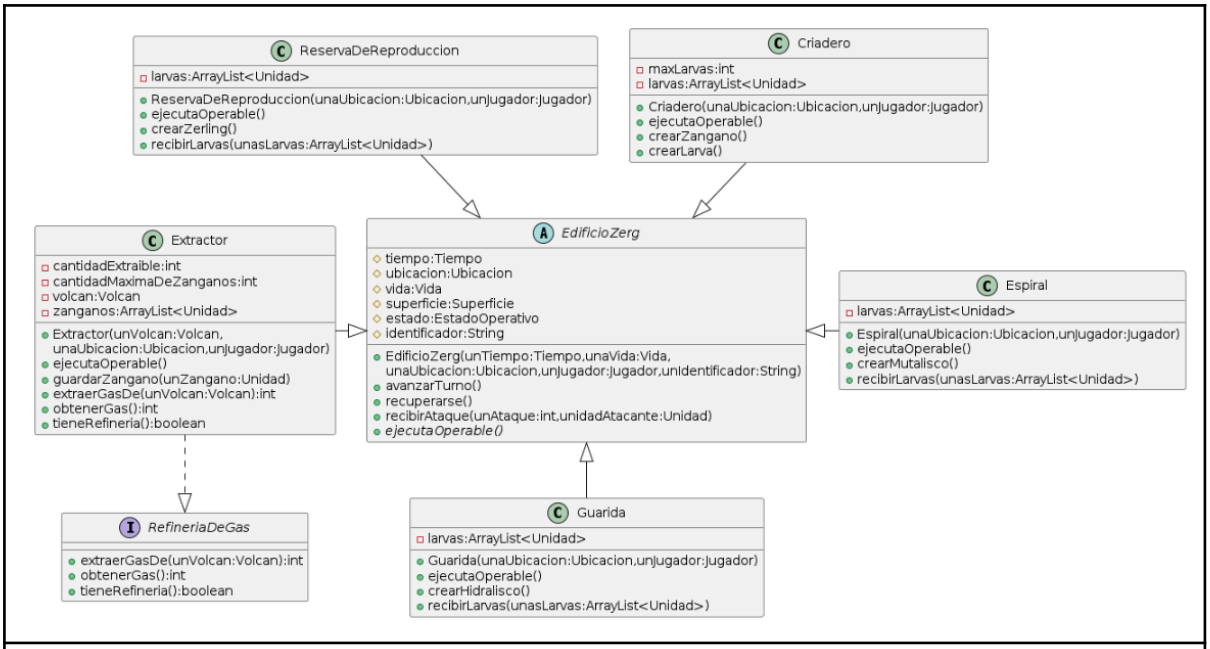
Relación entre Raza y sus clases hijas

Edificio



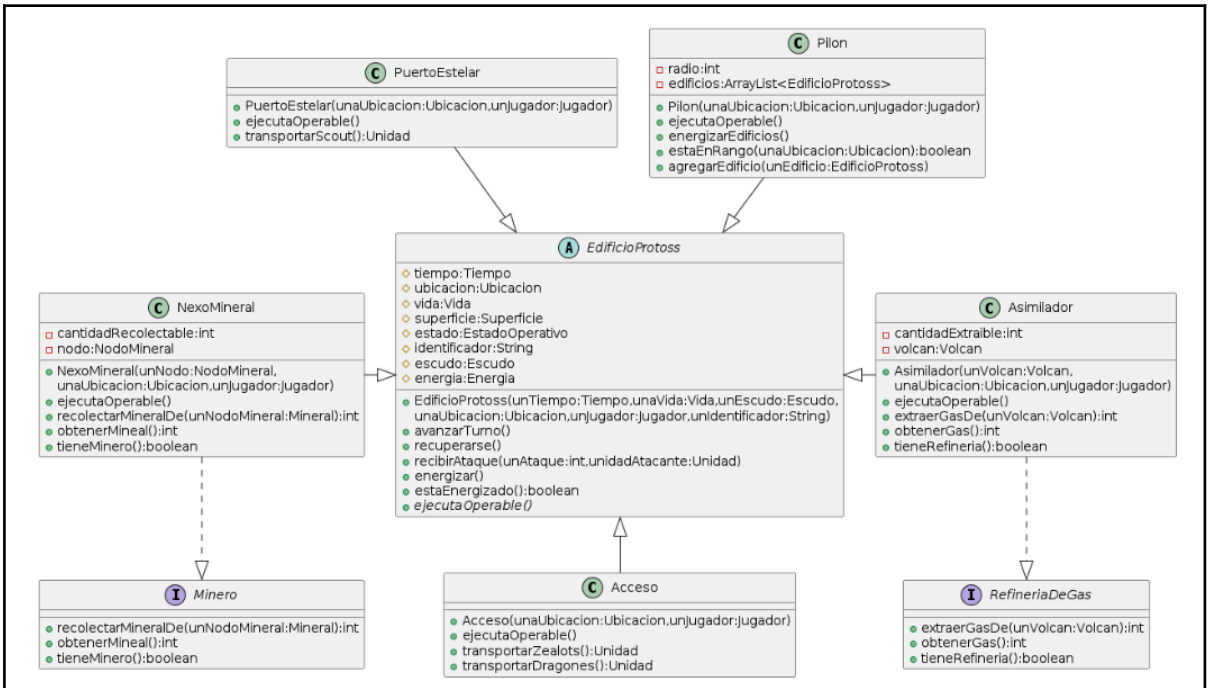
Relación entre Edificio y sus clases hijas

Edificio Zerg



Relación entre EdificioZerg y sus clases hijas

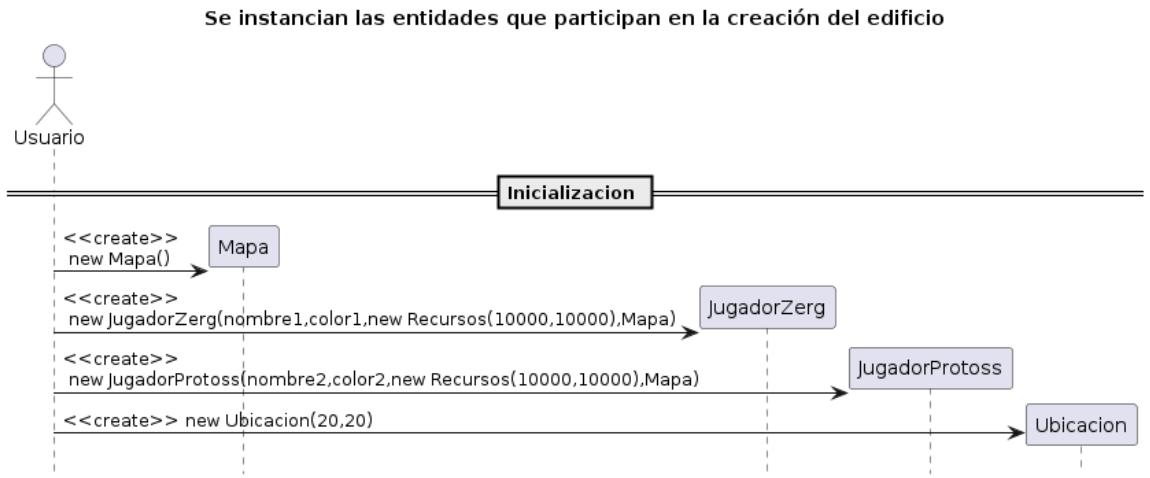
Edificio Protoss



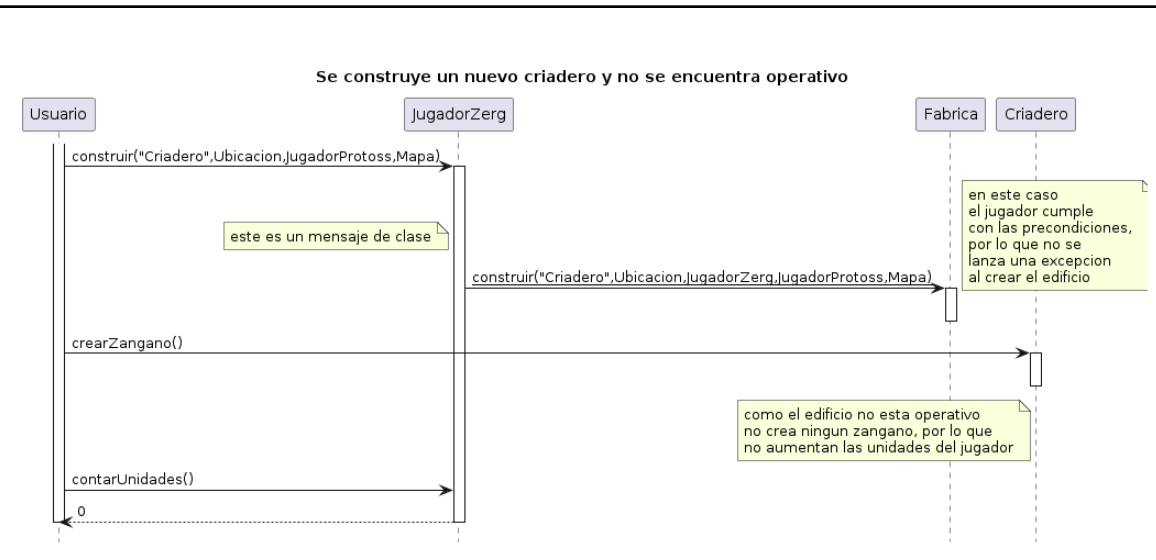
Relación entre EdificioProtoss y sus clases hijas

Diagramas de secuencia

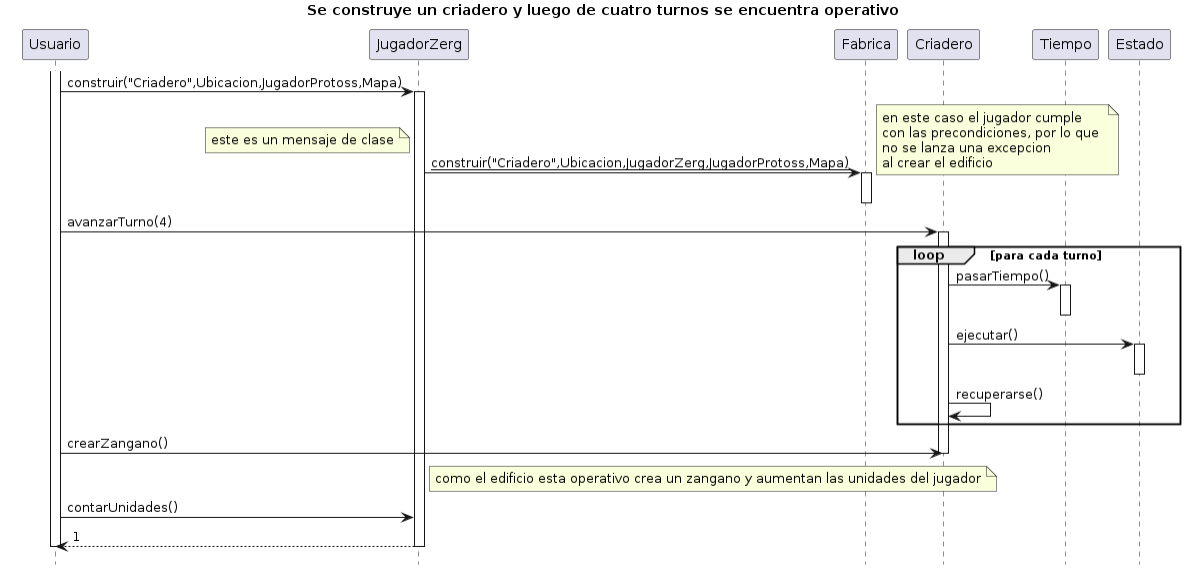
Caso de uso 2



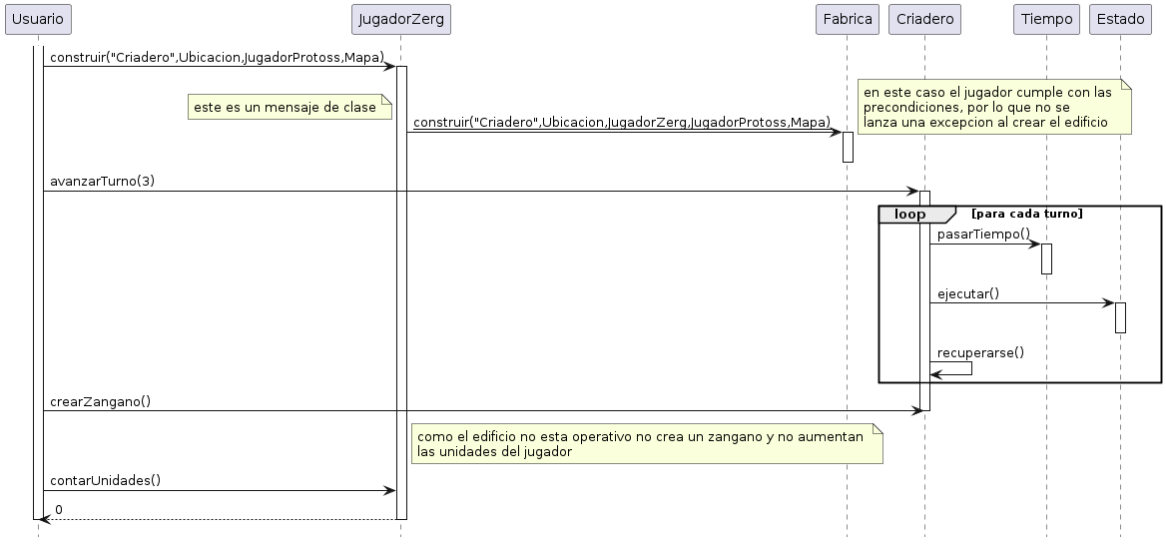
Inicialización de entidades



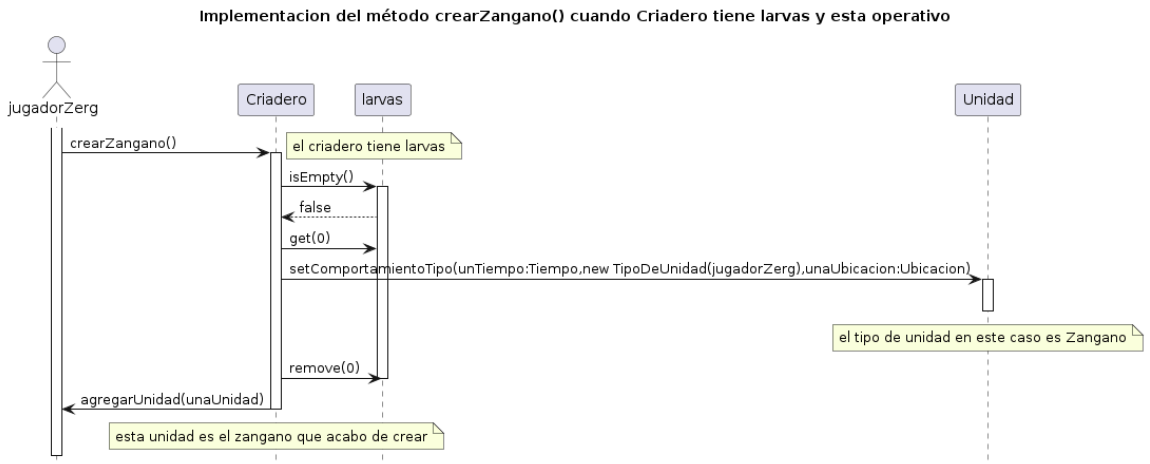
Nuevo Criadero



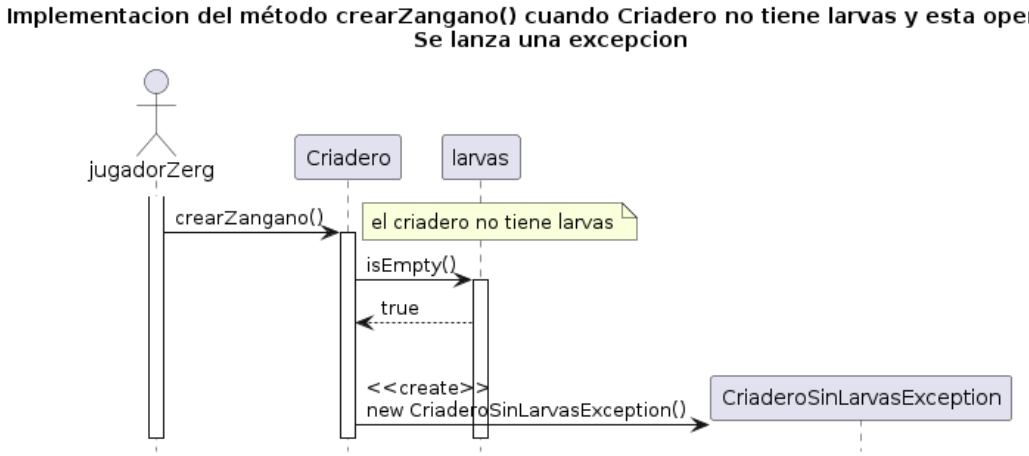
Criadero operativo luego de cuatro turnos



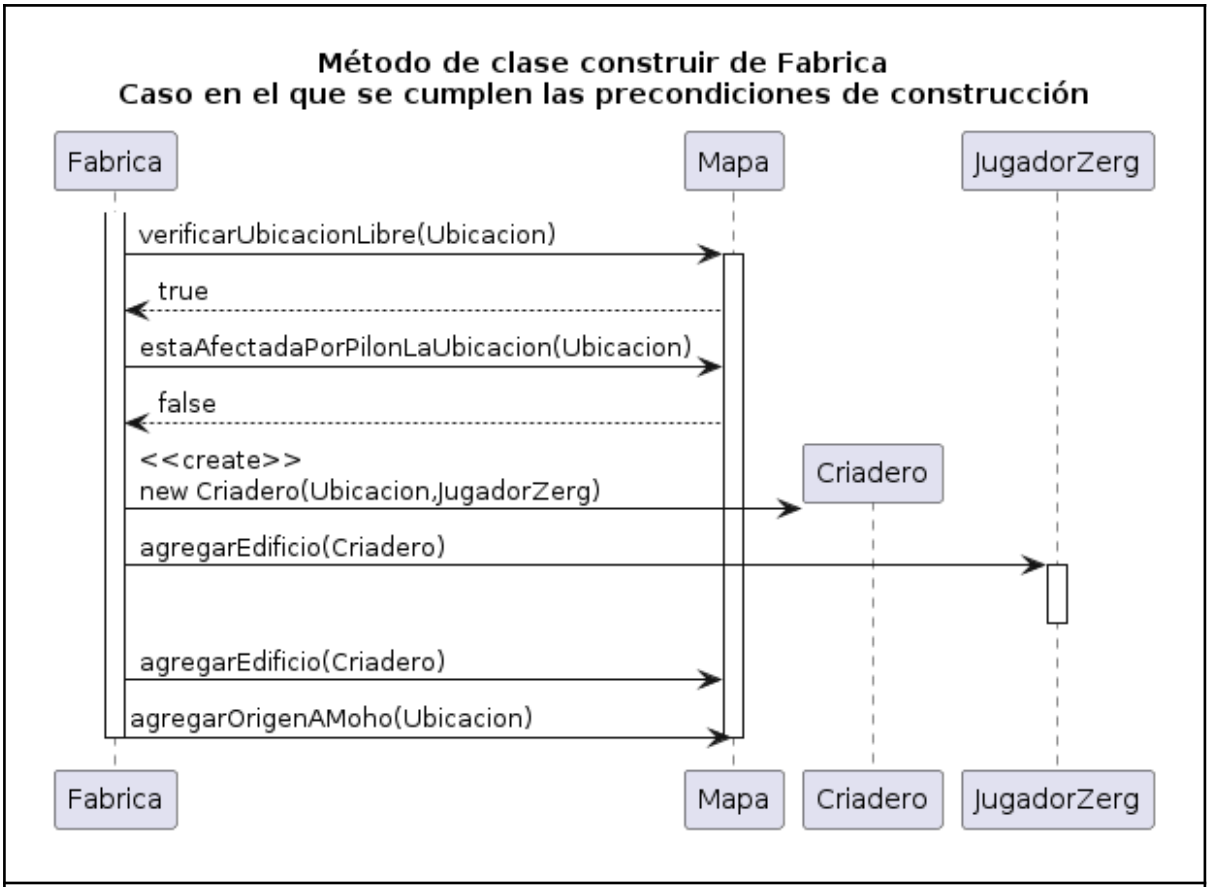
Criadero no operativo luego de tres turnos



Implementación de crearZangano()

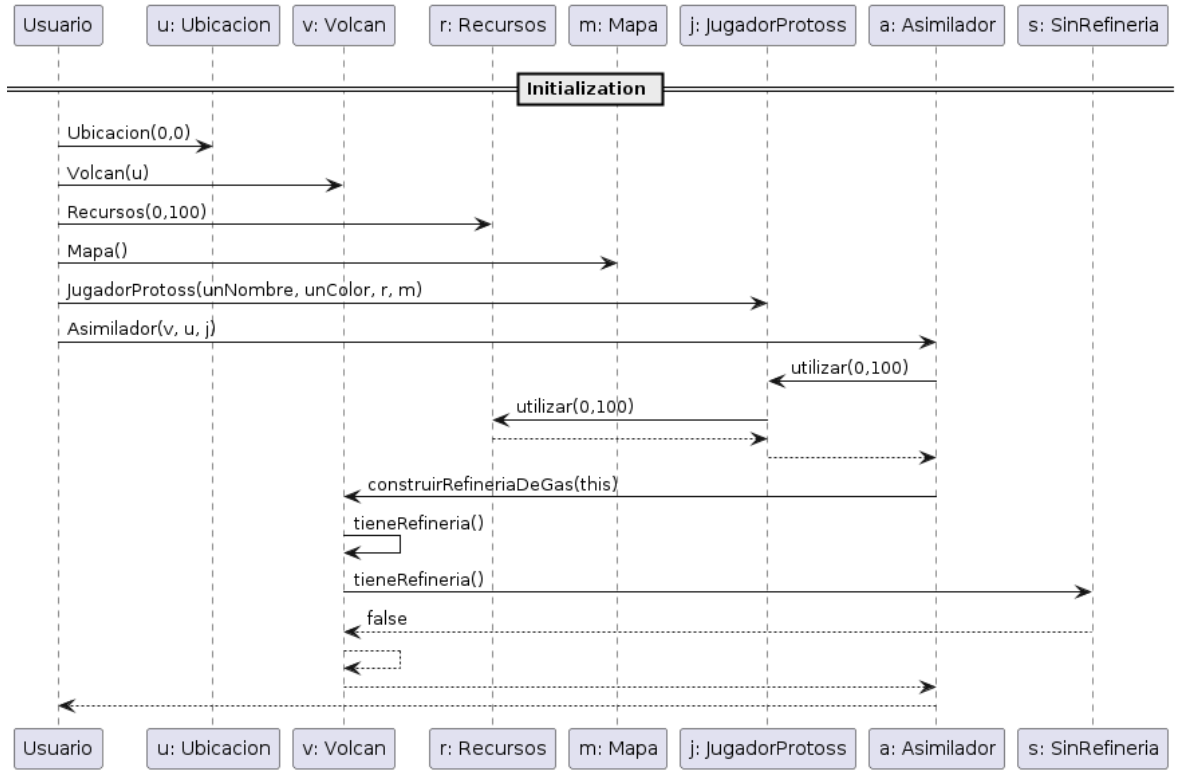


crearZangano() lanza una excepción

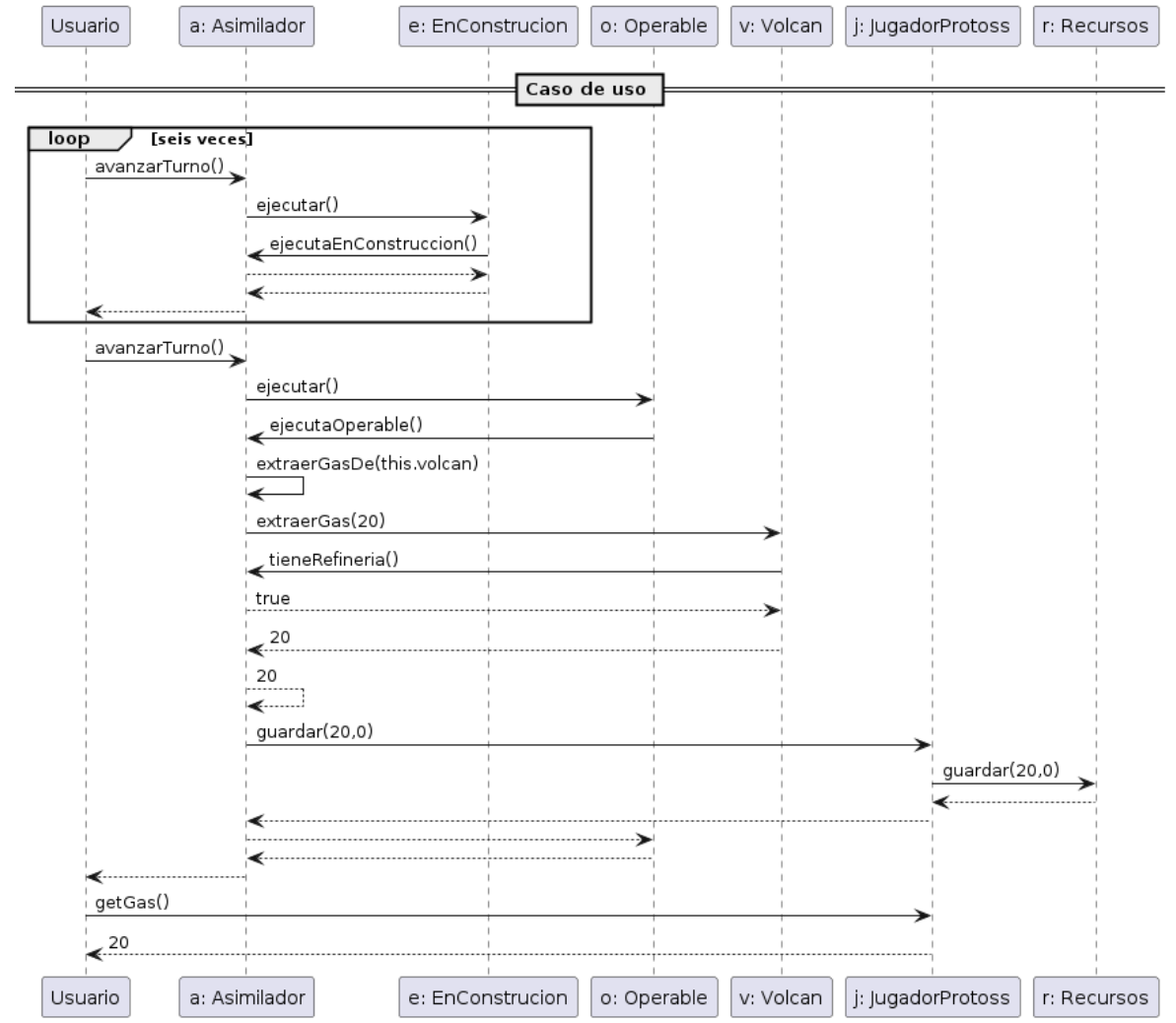


Método de clases construir de Fábrica para crear un Criadero

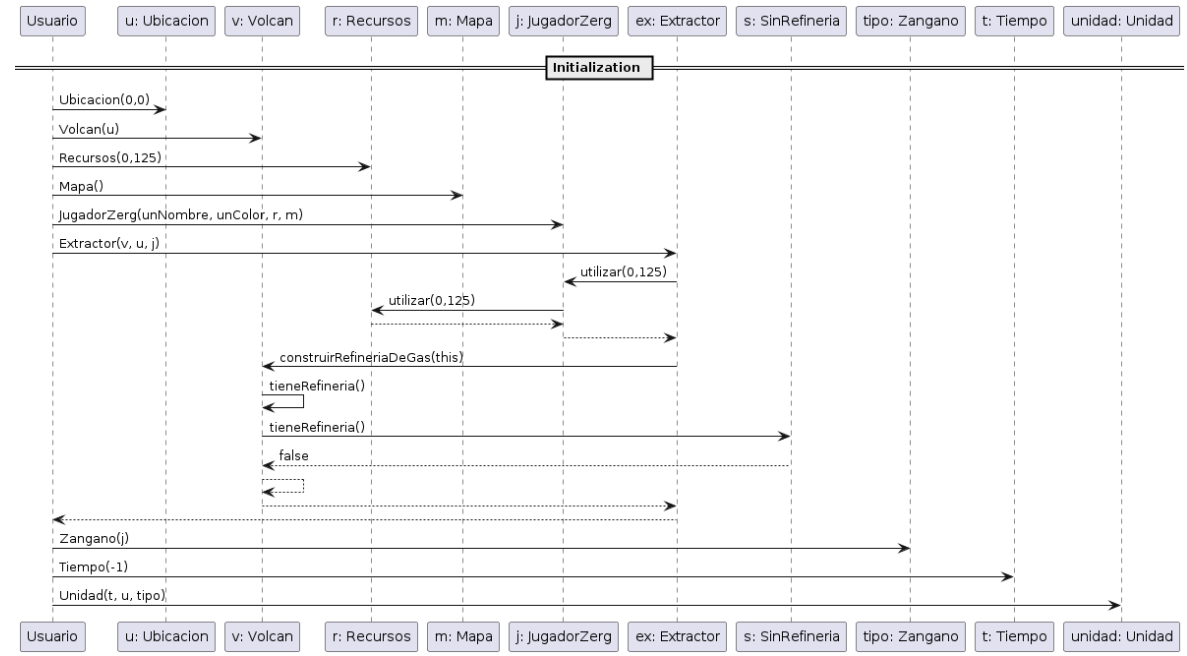
Caso de uso 4



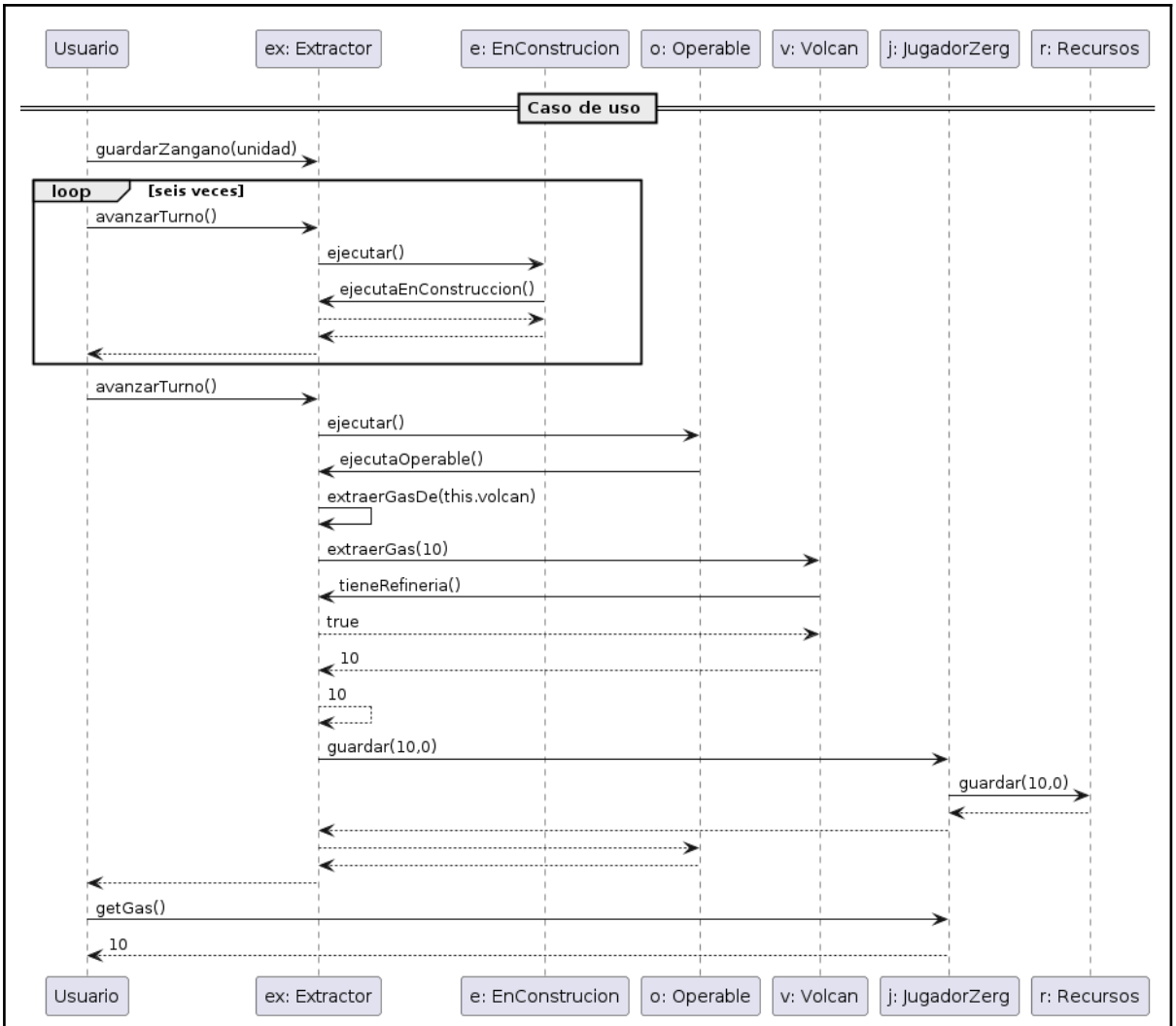
Caso de uso 4 - Asimilador. Inicialización.



Caso de uso 4 - Asimilador.

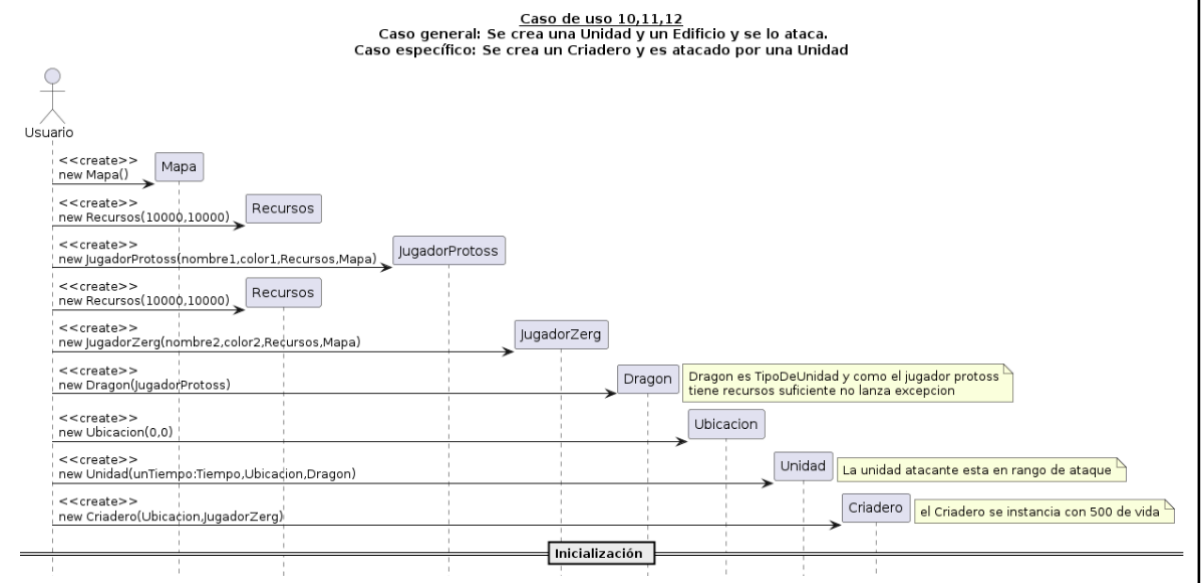


Caso de uso 4 - Extractor. Inicialización.

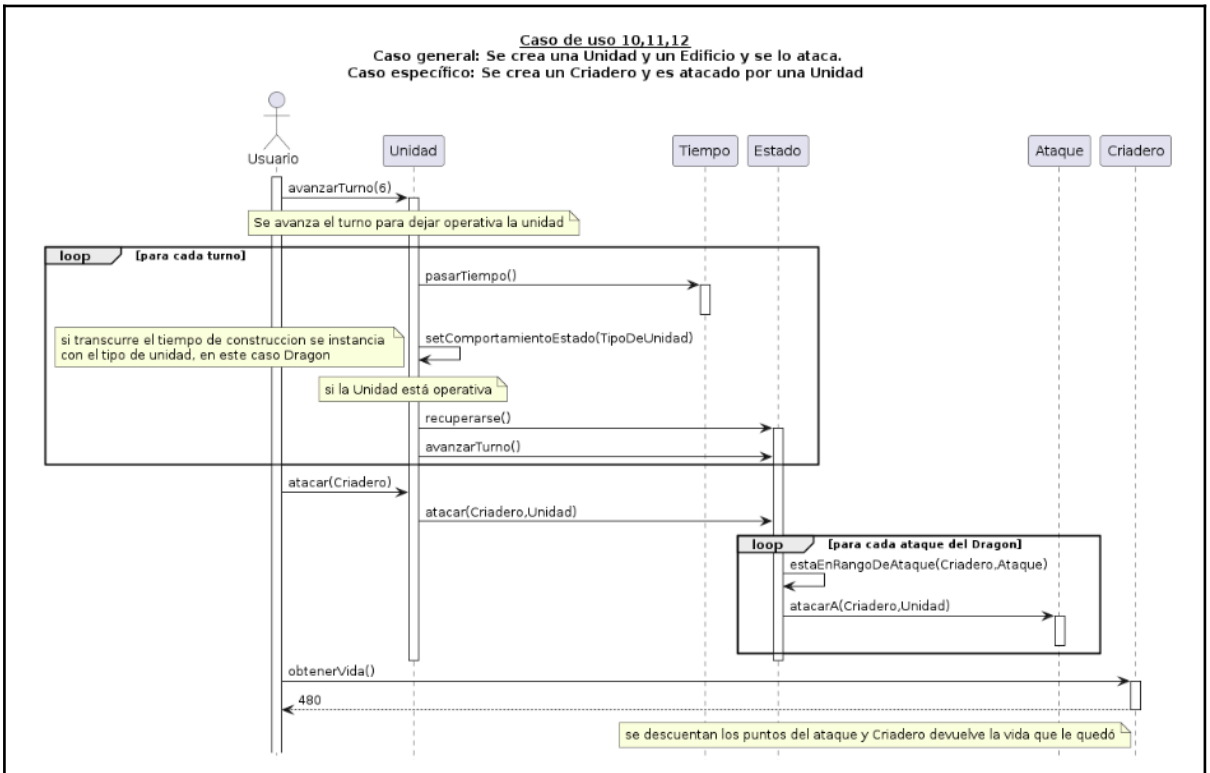


Caso de uso 4 - Extractor.

Caso de uso 10

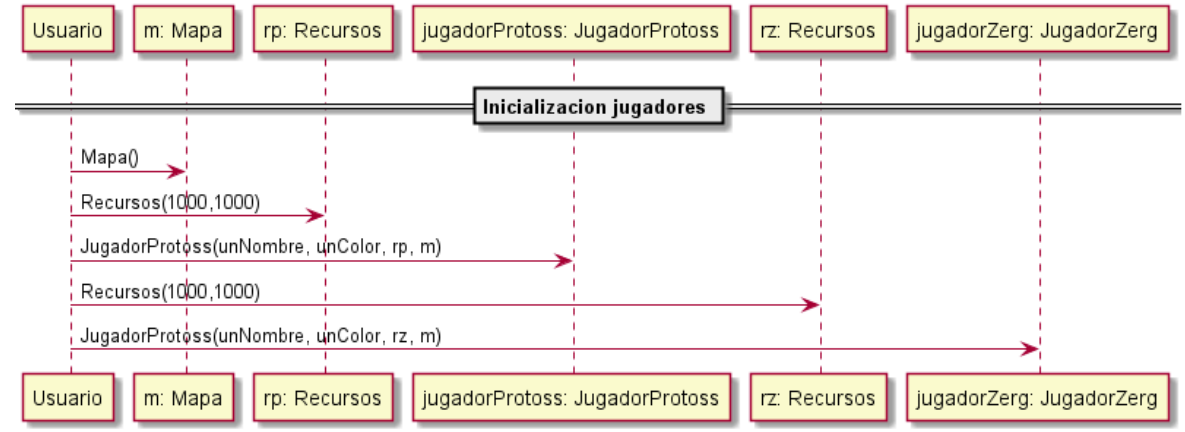


Inicialización caso de uso 10

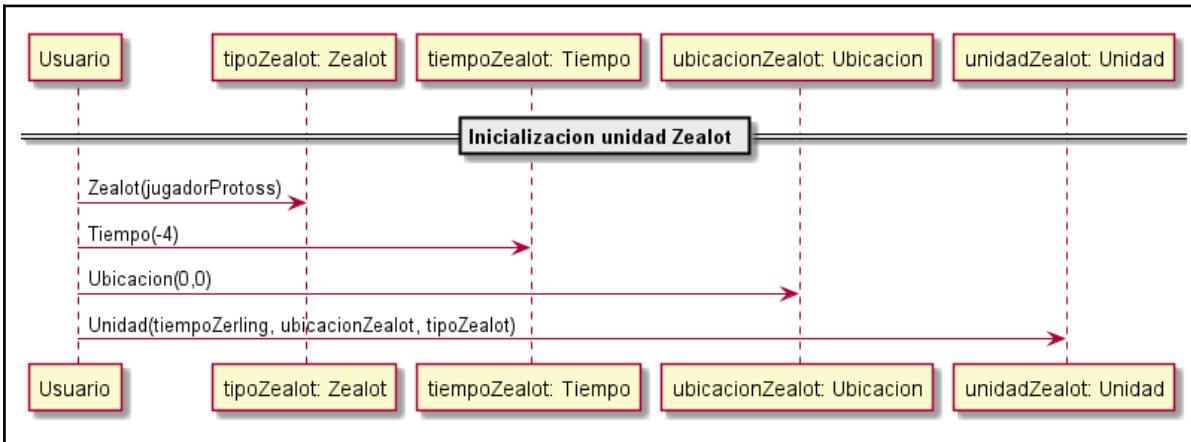


Ataque de una Unidad a un Criadero

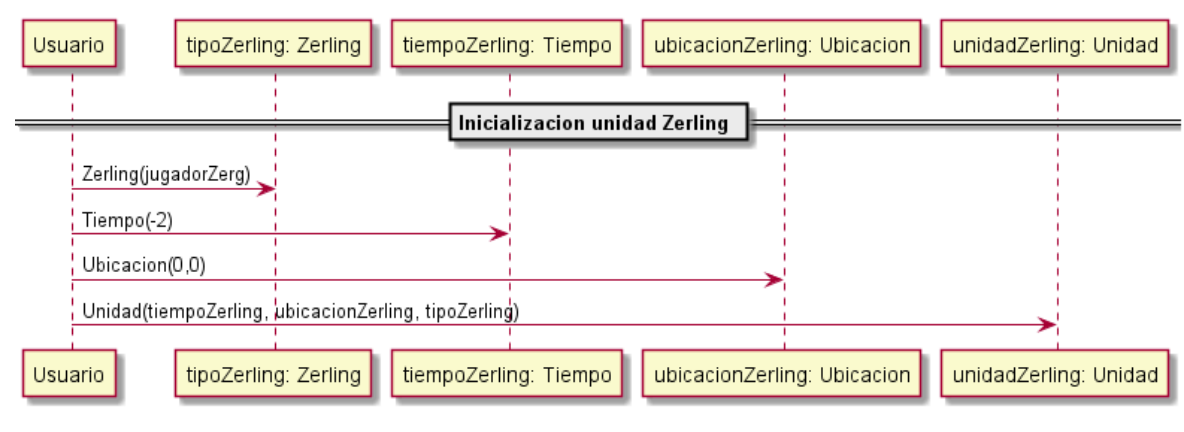
Caso de uso 18



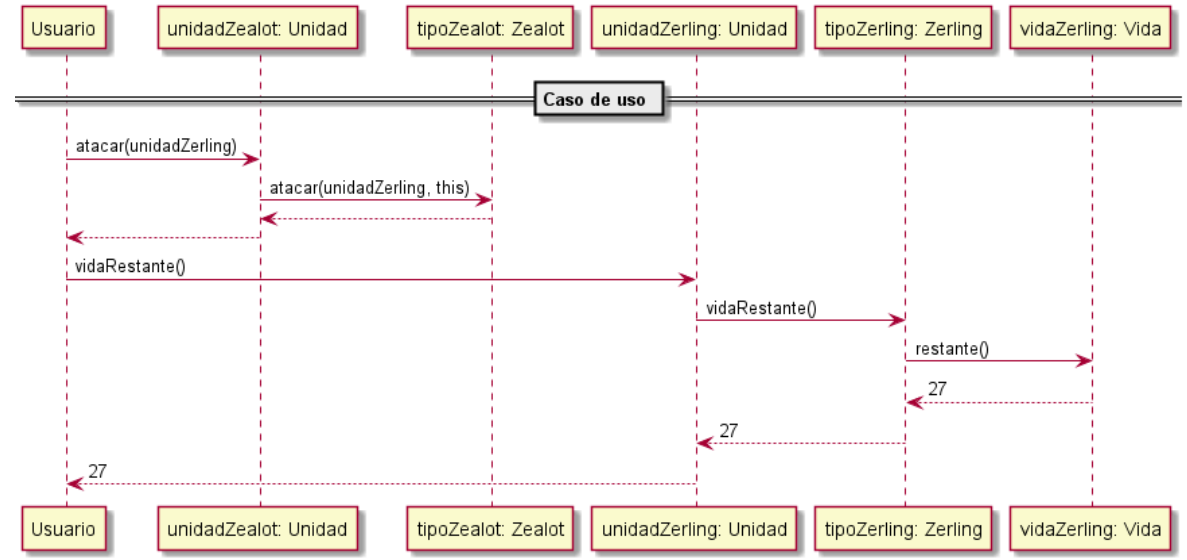
Caso de uso 18. Inicialización jugadores.



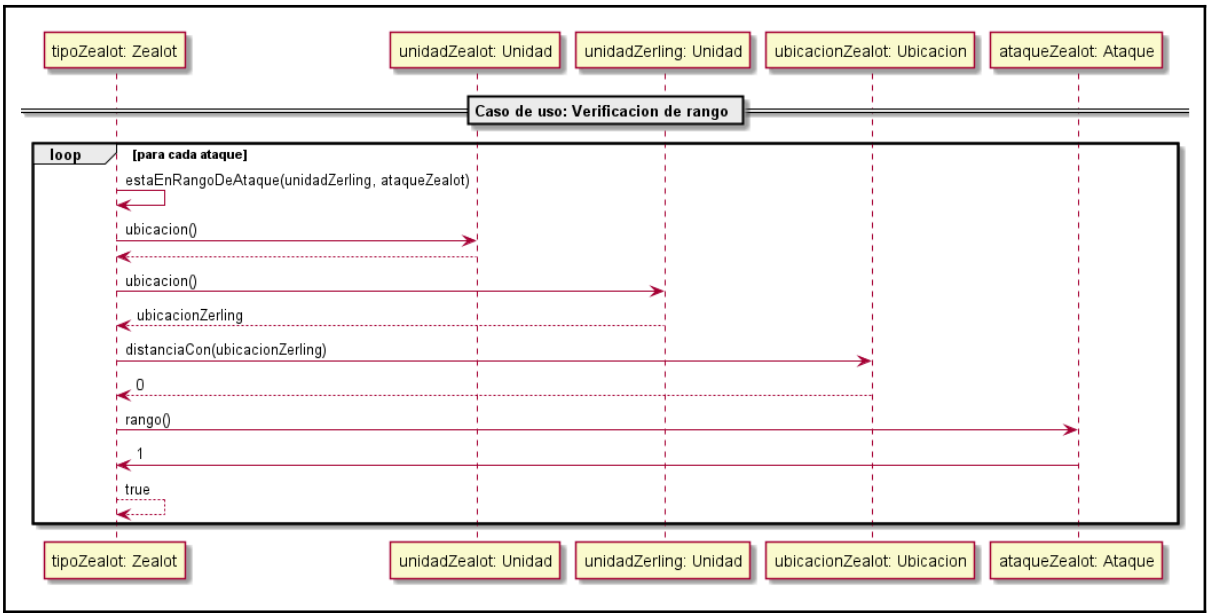
Caso de uso 18. Inicialización Zealot.



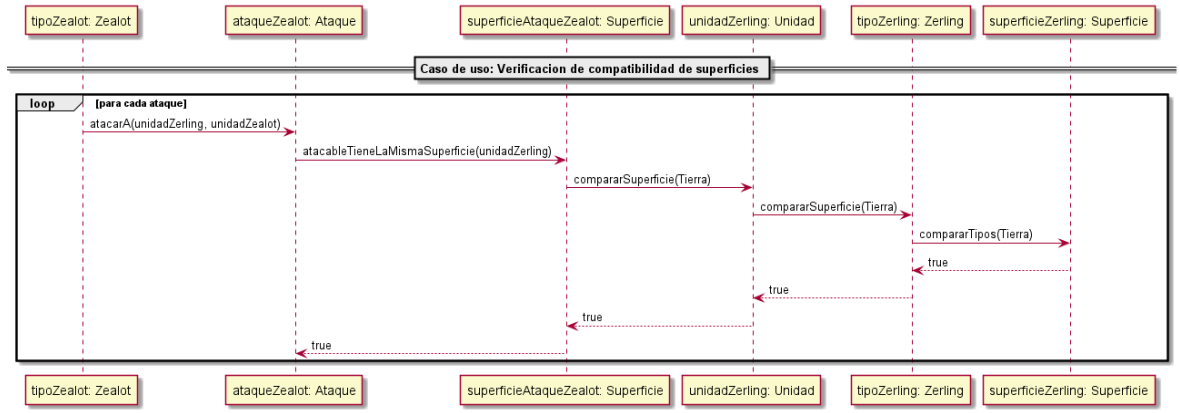
Caso de uso 18. Inicialización Zerling.



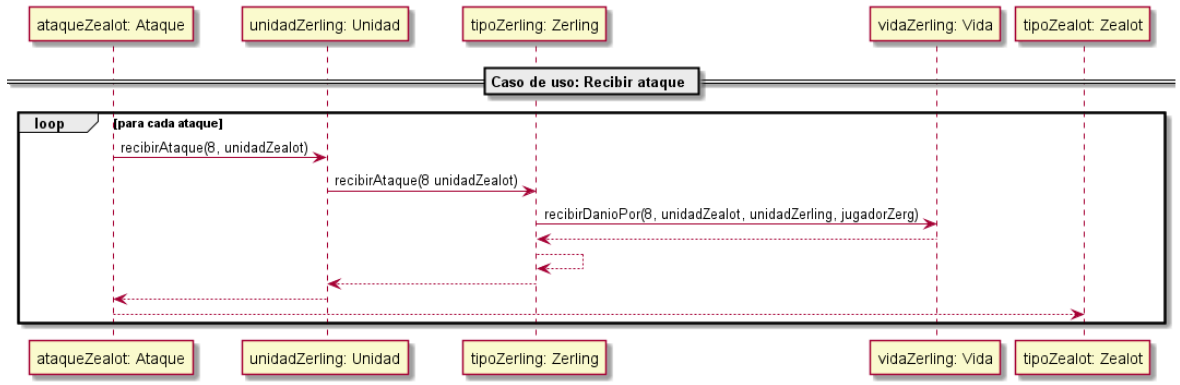
Caso de uso 18.



Caso de uso 18. Loop (verificación de rango).

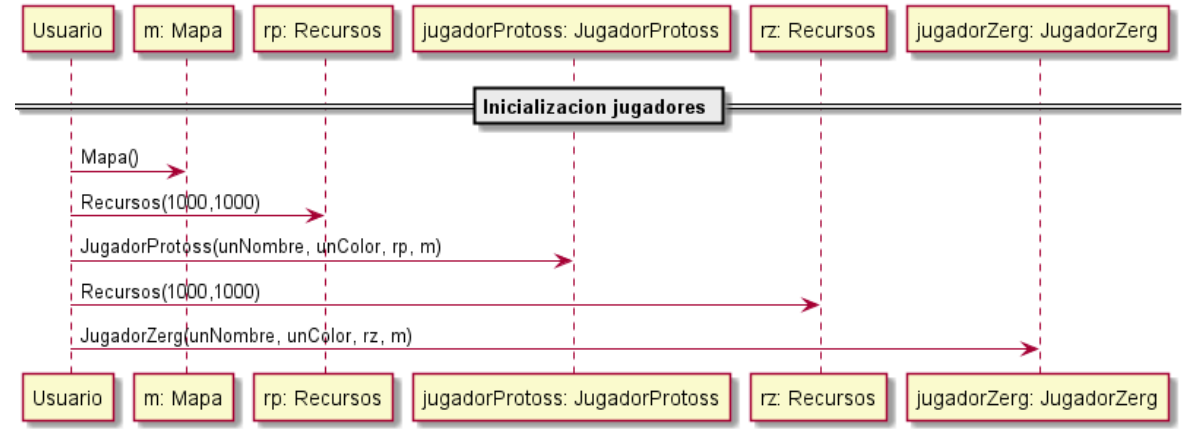


Caso de uso 18. Loop (verificación de compatibilidad de superficies).

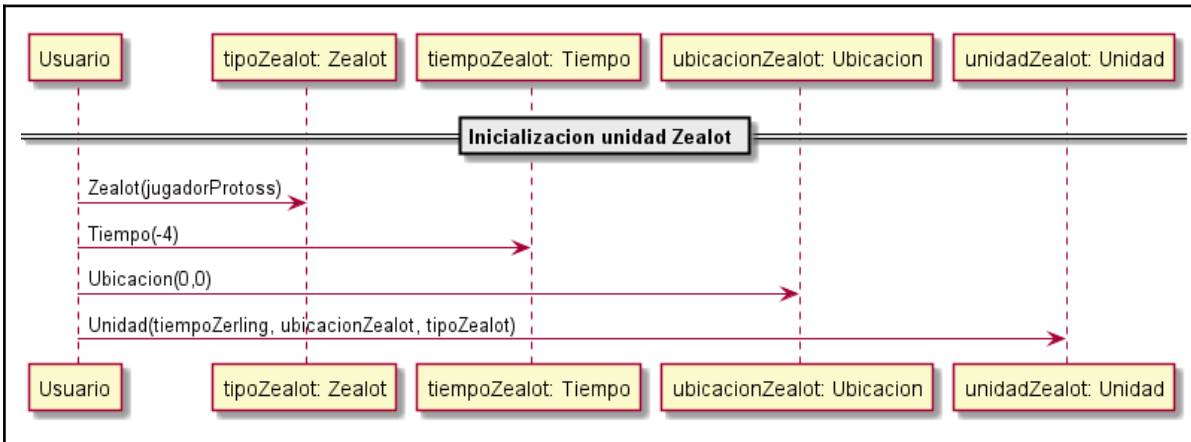


Caso de uso 18. Loop (recibir ataque).

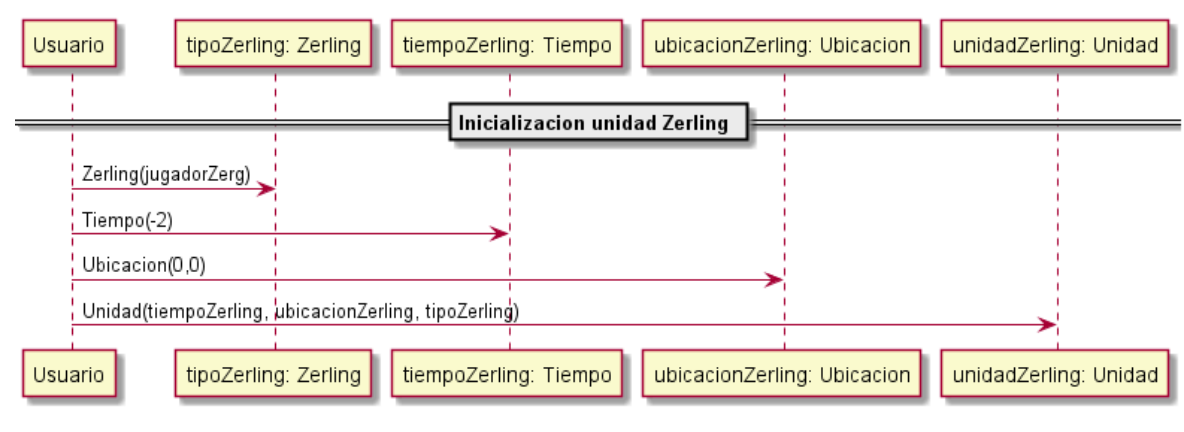
Caso de uso 28



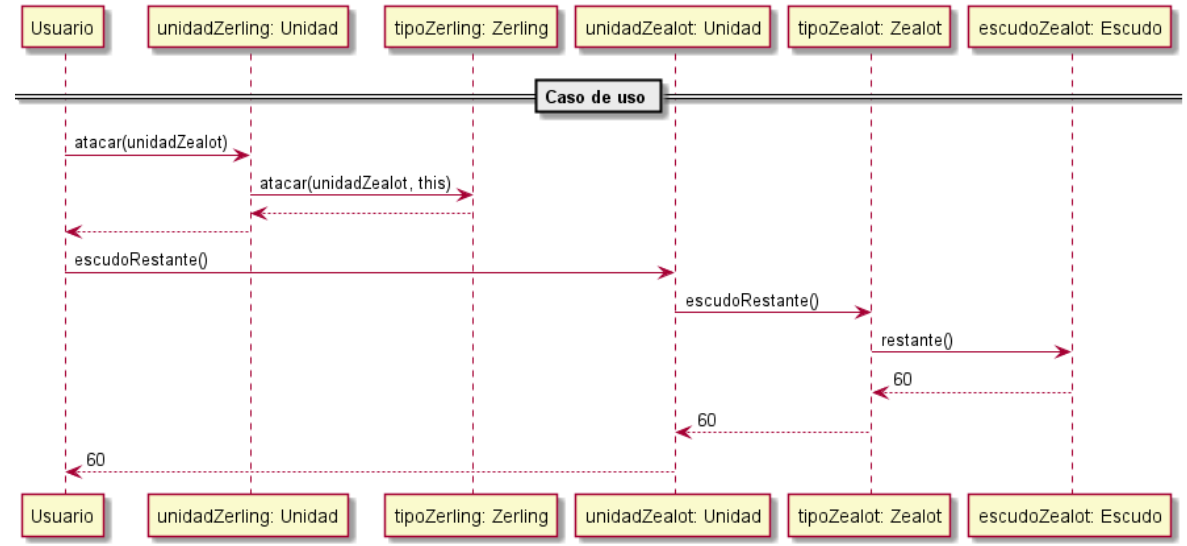
Caso de uso 28. Inicialización jugadores.



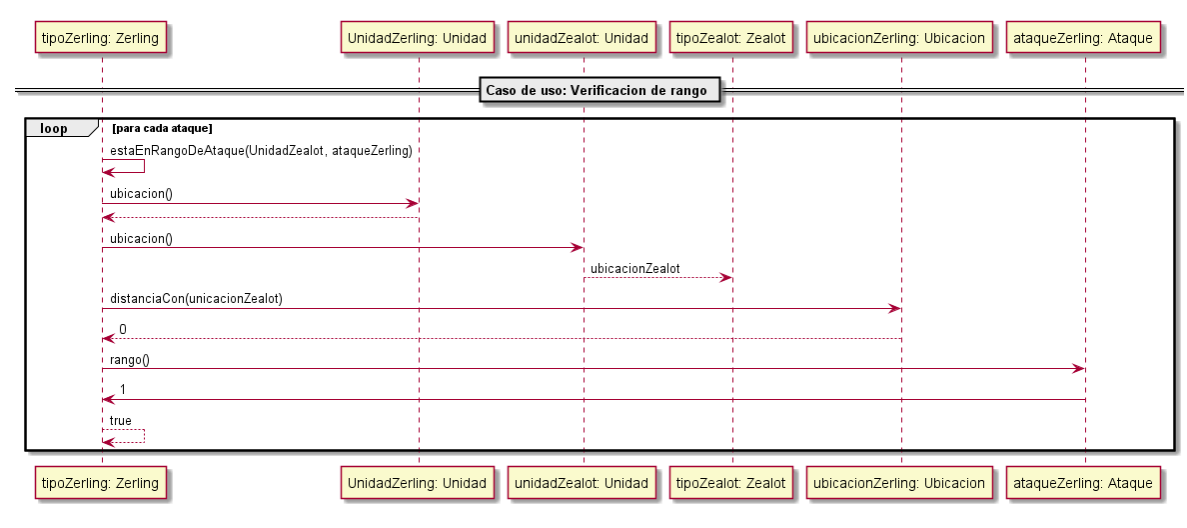
Caso de uso 28. Inicialización Zealot.



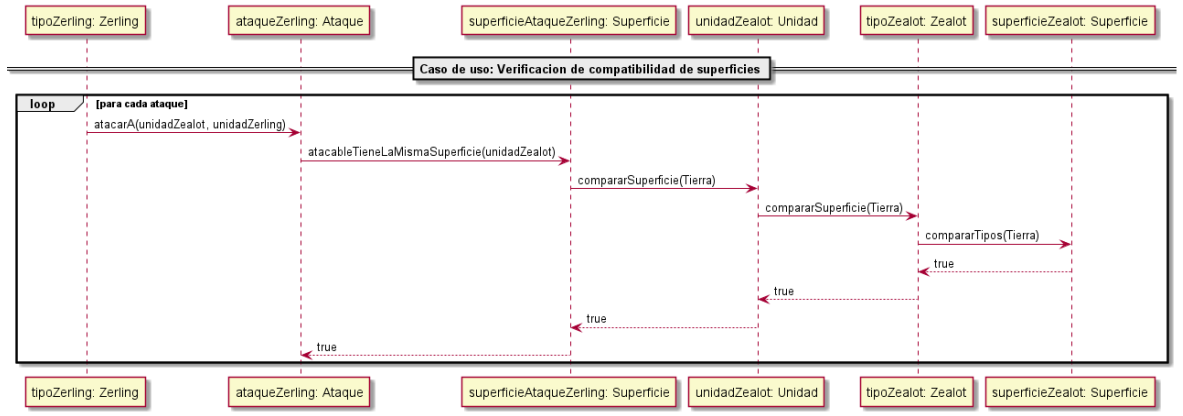
Caso de uso 28. Inicialización Zerling.



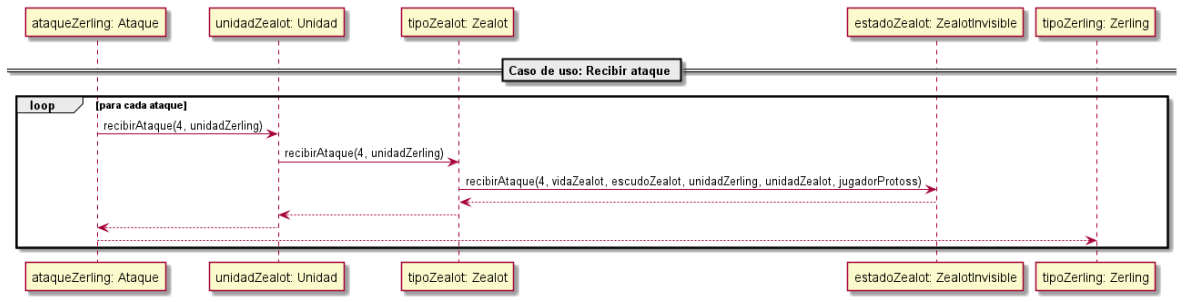
Caso de uso 28.



Caso de uso 28. Loop (verificación de rango).

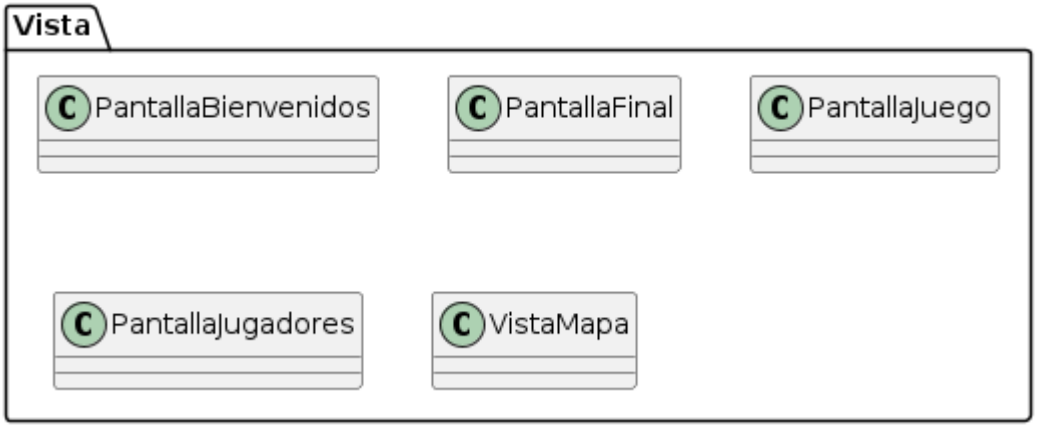


Caso de uso 28. Loop (verificación de compatibilidad de superficies).

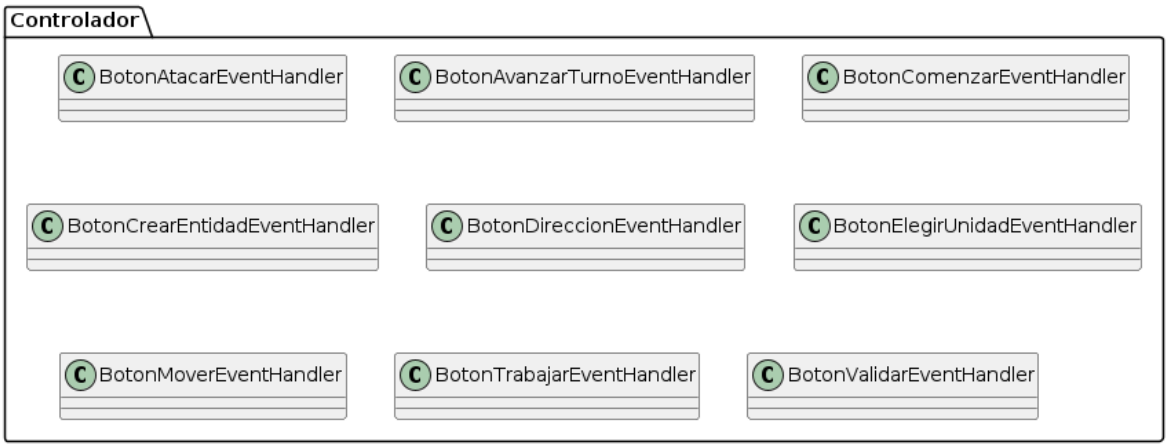


Caso de uso 28. Loop (recibir ataque).

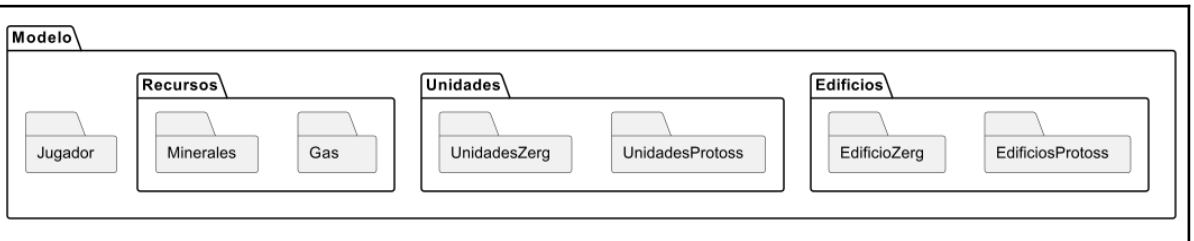
Diagrama de paquetes



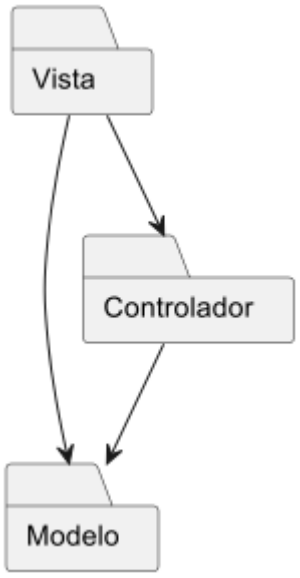
Vista



Controlador



Modelo



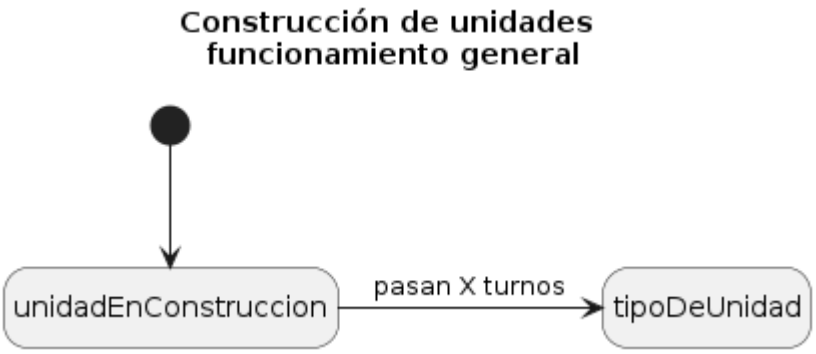
MVC

Diagramas de estado

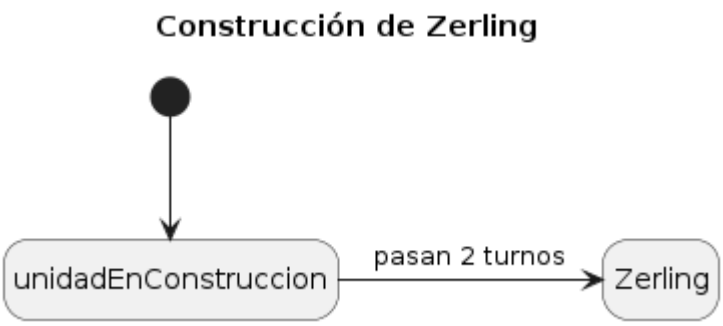
Haremos mención a los cambios de estado que consideramos más relevantes.

Cambios de estado en unidades

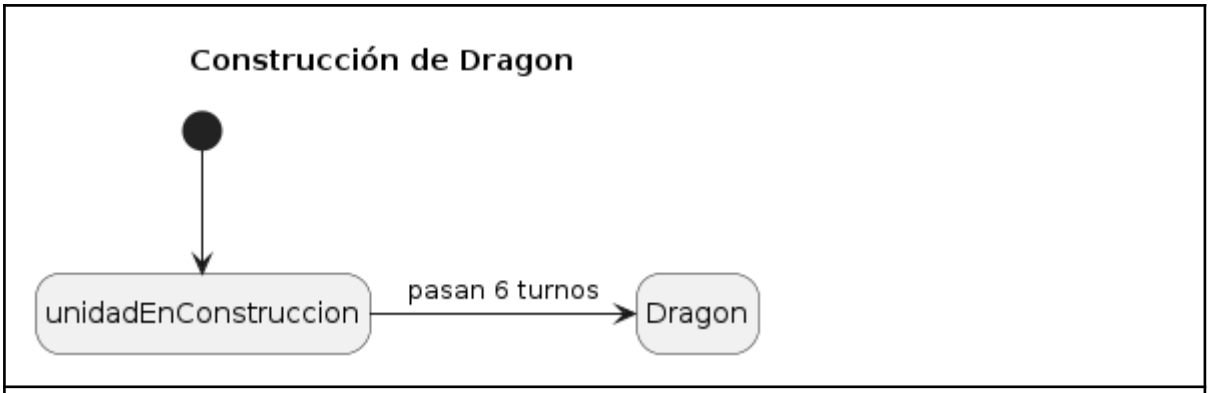
permitiendo modelar una transición entre no operabilidad y operabilidad. En general es como se ve en el siguiente diagrama. Ya que una unidad (por ej. un Zerling) requiere cierto tiempo para ser “construído”, al ser creado el objeto Zerling su estado es de “unidadEnConstrucción”. Luego de cierto tiempo su estado cambia a su “tipo”, por ej. Zerling.



Esquema general de transición de estado para una unidad



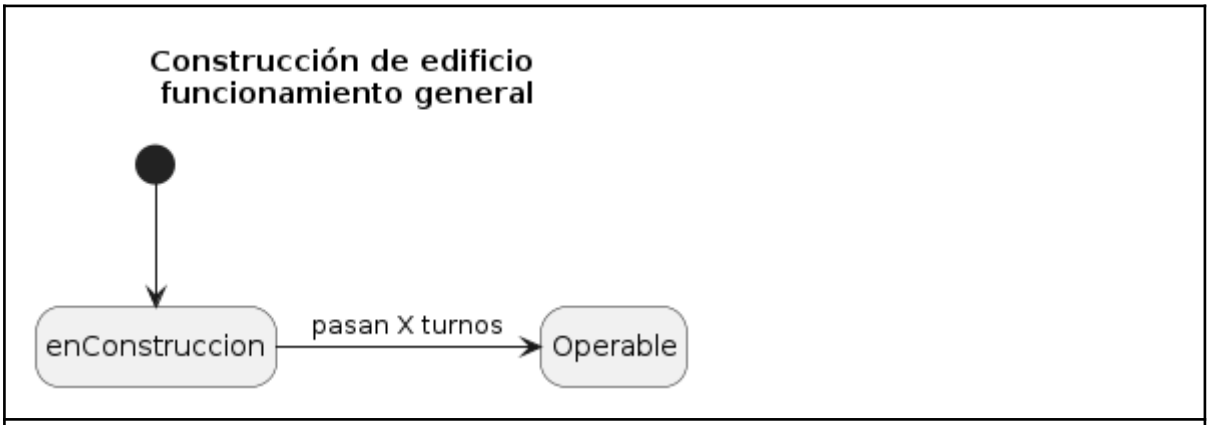
Caso concreto de transición de estado para una unidad Zerg



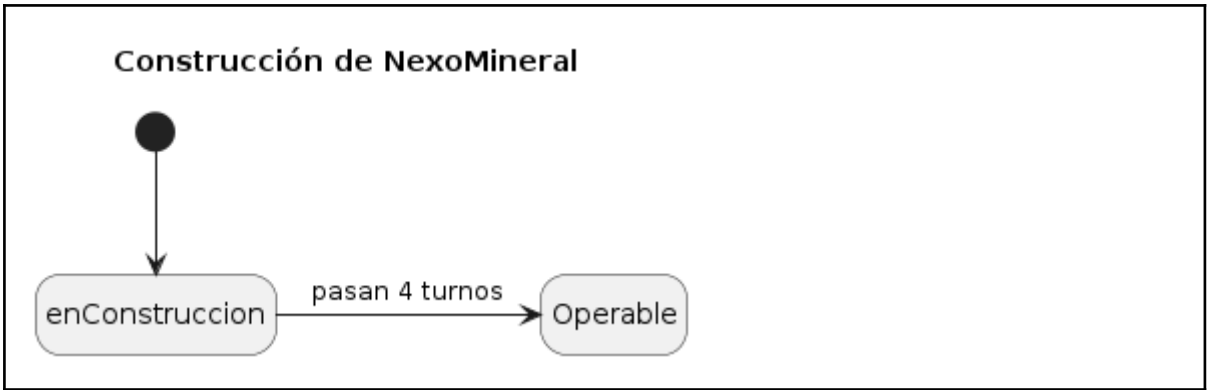
Caso concreto de transición de estado para una unidad Protoss

Cambios de estado en edificios

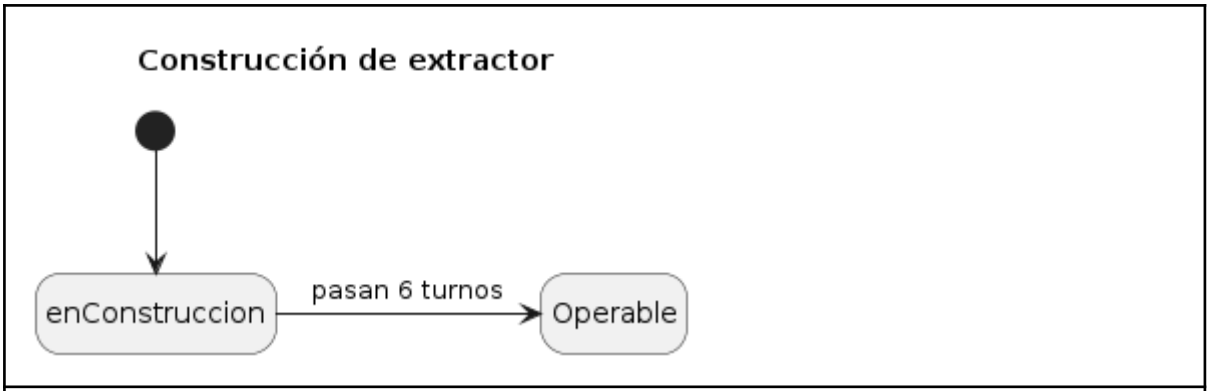
Similar al caso de las unidades.



Esquema general de transición de estado para un edificio



Caso concreto de transición de estado para un edificio Protoss

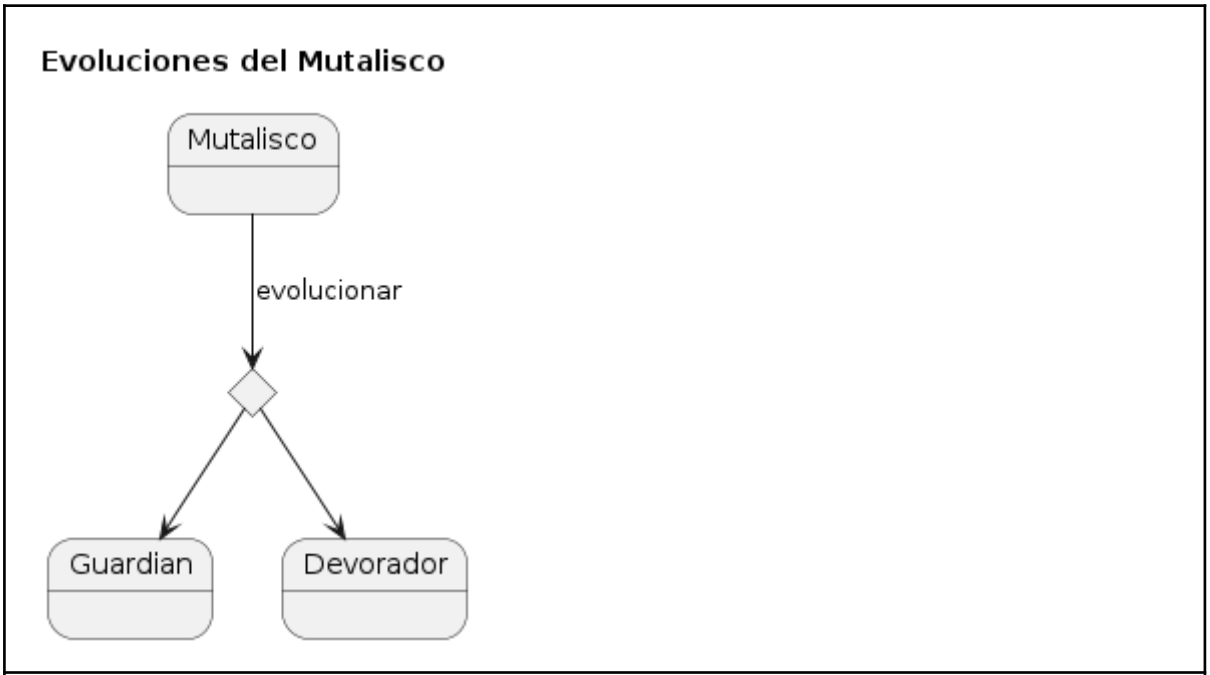


Caso concreto de transición de estado para un edificio Zerg

Casos particulares

Mutalisco

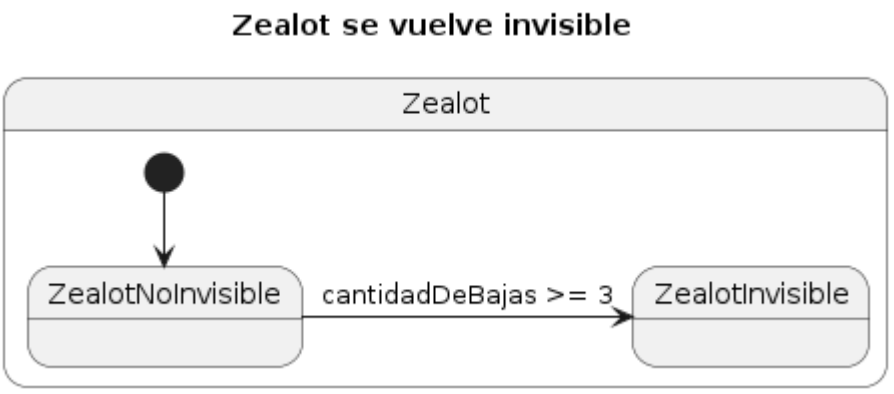
Una transición de estado como la que experimenta el Mutalisco al evolucionar. En caso de contar con los recursos suficientes el jugador podrá optar por crear un Guardián o un Devorador.



Un mutalisco puede cambiar a Guardián o Devorador

Zealot

Cambia su visibilidad luego de haber aniquilado a tres unidades enemigas.



Un Zealot se vuelve invisible al eliminar 3 enemigos

Detalles de implementación

Deben detallar/explicar qué estrategias utilizaron para resolver todos los puntos más conflictivos del trabajo práctico. Justificar el uso de herencia vs. delegación, mencionar que principio de diseño aplicaron en qué caso y mencionar qué patrones de diseño fueron utilizados y por qué motivos.

- La construcción de edificios y unidades comienza en el valor negativo correspondiente. En cada turno irá aumentando su valor y al llegar a cero se considera la entidad como construida.

- Recolección de recursos: Para resolver este punto del trabajo práctico hemos decidido utilizar delegación y aplicamos el principio de inversión de dependencias. Decidimos que tanto Asimilador como Extractor implementen una interfaz que llamamos RefineriaDeGas para que puedan entender los mismos mensajes relacionados a la recolección de gas y de esta forma que en el Volcán solo se pueda construir una refinería de gas. Además, de esta forma se aplica el principio de inversión de dependencias ya que tenemos que Volcán va a depender de la abstracción RefineriaDeGas y no de una implementación como podría ser Extractor o Asimilador. A continuación se detalla un poco como es la implementación de este punto del trabajo práctico. Cuando las refinerías de gas reciben el mensaje para avanzar turno, se ejecuta el método de "ejecutaOperable()" y en la implementación de este método lo que sucede es que le delegamos al volcán la responsabilidad de extraer el gas mediante el método "extraerGas()" y le pasamos la cantidad extraíble. El objeto Volcán lo que hace dentro de este método es hacer las verificaciones correspondientes, descontarse la cantidad de gas y devolver la cantidad pedida para luego delegarle al jugador la responsabilidad de guardarse la cantidad de gas correspondiente. Esta idea es análoga para la recolección de mineral mediante NexoMineral y Zangano.

- Ataque: Para resolver este punto del trabajo práctico hemos decidido utilizar delegación mediante la clase Ataque. Aquellas entidades que puedan ser atacadas implementan la interfaz que llamamos Atacable y aquellas entidades que pueden atacar implementan la interfaz que llamamos Atacante. La idea que tuvimos al considerar estas interfaces fue aplicar el principio de segregación de la interfaz ya que nos aseguramos de que el cliente no sea forzado a depender de métodos que no utiliza. Por ejemplo, el tipo de unidad Zángano implementa la interfaz Atacable pero no la Atacante y entonces no sabe atacar. El objeto Unidad tiene una referencia al objeto que representa su tipo de unidad dentro y cada tipo de unidad tiene dentro una lista con referencias a sus ataques. Cuando se le dice a la unidad que implementa la interfaz Atacante que ataque, esta delega en su tipo de unidad y su tipo de unidad delega en sus ataques. Lo que se hace es iterar la lista y por cada ataque se chequea que esté en el rango correcto y luego que la superficie del ataque sea compatible con la unidad que está siendo atacada. Para chequear el tipo de superficies, la clase Ataque delega en la clase Superficie la responsabilidad de hacer las comparaciones correspondientes. Luego, una vez que se cumplen los chequeos correspondientes se delega a la unidad Atacable la responsabilidad de recibir el daño. Por último, mediante varias delegaciones más en el medio, el tipo de unidad termina delegando al objeto Vida la responsabilidad de efectuar el daño del ataque. ¿Por qué los tipos de unidades tienen una referencia al objeto Unidad?. Esta decisión en la implementación la tomamos debido a que cuando la vida de la unidad llegará a cero, de alguna forma teníamos que tener una referencia a la unidad sin vida para enviarle un mensaje al jugador dueño de dicha unidad y utilizar dicha referencia para eliminarlo de su lista de unidades. Luego, tener dicha referencia dentro de los tipos de unidades nos ayudaba a no tener que estar pasando esa referencia por los parámetros de los métodos.

- Unidades: Para resolver este punto del trabajo práctico hemos decidido utilizar principalmente delegación aplicando el patrón State. Decidimos utilizar el patrón State debido a que necesitábamos que la clase Unidad altere su comportamiento en tiempo de ejecución. Es decir, necesitábamos que de alguna forma el objeto cambiará su clase. La idea es que el objeto Unidad almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto (atacar, recibir daño, etc). En el contexto de este trabajo práctico, inicialmente el objeto Unidad almacena una referencia al objeto UnidadEnConstruccion que representa su estado en construcción. Luego de avanzar una X cantidad de turnos lo que sucede es que su estado actual se actualiza y cambia al tipo de unidad. Notemos que para la transición de un estado a otro se sustituye el objeto de estado activo por otro objeto que represente el nuevo estado. Esto solo es posible si todas las clases de estado siguen la misma interfaz y por eso mismo los tipos de unidad (Zangano, Zerling, Scout, Drago, etc) implementan la interfaz TipoDeUnidad. Notemos que al utilizar este patrón de diseño tenemos que se aplican los siguientes principios de diseño: *Principio de responsabilidad única. Organiza el código relacionado con estados particulares en clases separadas. *Principio de abierto/cerrado. Introduce nuevos estados sin cambiar clases de estado existentes o la clase Unidad.

- Las unidades comienzan a moverse en la dirección derecha y al cambiar de dirección lo hacen en sentido horario. Necesitan un mapa para moverse, ya que este hace las verificaciones correspondientes a la ubicación a la cual se quiere mover la unidad.

- Mapa: El mapa es el contenedor y gestor de otras entidades que producen cambios en el modelo cuando el jugador realiza alguna acción. Si el jugador quiere crear un edificio o mover una unidad, es el mapa el encargado de verificar que dicho edificio pueda ser construido en una ubicación, que la unidad se pueda mover a una ubicación, que la ubicación este en el mapa, también se encarga de revelar a las unidades iterando la lista de amos supremos, que los pilones energizan los edificios protoss, agregar un origen desde el cual se expande el moho cada vez que se crea un criadero, etc. En un comienzo se pensó en usar el patrón de diseño “Singleton” para el mapa(algo bastante común) pero se decidió dejar el patrón de lado y que cada jugador tenga una referencia a un único mapa que se instancia antes de empezar el juego.

- Jugador: Es una clase abstracta de la cual heredan las clases JugadorZerg y JugadorProtoss. En este caso se usa la herencia para favorecer la reutilización de código, ya que solo en casos muy puntuales difieren en implementación, y además ambos cumplen con la relación “es un” que caracteriza a la herencia. Ambos tipos de jugadores tienen una colección de edificios, una colección de unidades y un mapa con el cual van a interactuar durante el desarrollo del juego.

- Fábrica: El objetivo de la clase Fábrica es servir de gestor a la hora de crear edificios y unidades correspondientes. En un inicio se pensaba usar el patrón de diseño “Factory Method” para crear los edificios o unidades que se crean necesarios, pero los cambios en implementación de otras clases hicieron que se opte por hacer una clase común con un método estático. La ventaja de la implementación en el modelo es en primer lugar la utilización de polimorfismo, ya que cualquier jugador entiende el mensaje construir y ambos le delegan la “construcción” a Fábrica, y en segundo lugar es que el concepto de creación vive en una sola entidad, entonces Fabrica hace las verificaciones y delegaciones correspondientes.

- Edificio: Edificio es una clase abstracta hereda de la clase abstracta Raza, pues edificio es una entidad que se puede atacar y necesita de un tiempo y una ubicación. De Edificio heredan las clases abstractas EdificioZerg y EdificioProtoss, los cuales serán usados por los jugadores correspondientes. En la implementación interna de cada edificio se nos presentó el problema de cómo hacer para que estén operativos transcurridos el tiempo correspondiente, y una primera aproximación fue la de implementar el patrón de diseño “Strategy” e ir cambiando el estado de un edificio en tiempo de ejecución. El inconveniente con esto es que aumentaba demasiado la cantidad de clases en el modelo, por lo que se optó por usar un pseudo patrón “Command”. Lo que hicimos es tratar de identificar qué acción repiten todos los edificios al avanzar los turnos y los agrupamos en un método ejecutar que va a estar dentro de un Estado que puede ser operativo o no operativo. Ahora lo único que debíamos hacer es setear el estado correspondiente una vez transcurrido el tiempo, y ahora cada edificio tiene sus propias acciones. El estado siempre utiliza el método ejecutar al avanzar turno, si el estado es no operativo, lo que hace es chequear que el tiempo sea el adecuado para cambiar al estado operativo, y si el estado es operativo, realiza las acciones específicas de un edificio. Un ejemplo concreto es el Criadero, que hereda de EdificioZerg, su estado operativo lo que hace en cada turno es crear una larva. Ahora, ¿Por qué es un pseudo patrón? La respuesta es que en un principio, todos los edificios realizaban alguna acción específica, pero el desarrollo del modelo y los cambios, hacían que haya conflicto con las verificaciones que debían realizar otras clases en cada turno. Ahora el estado de un edificio llama a ejecutar, pero el estado operativo muchas veces no llama a ningún método específico del edificio en cuestión, por lo que no termina de ser un patrón Command en toda regla. La mayoría de esos llamados se hacen desde afuera de la clase.
  Por último, los detalles que promueven la programación por diferencia son la implementación de un escudo y energía en los edificios protoss, en casi todo lo demás se reutiliza el código de la clase madre.

Excepciones

Explicar las excepciones creadas, con qué fin fueron creadas y cómo y dónde se las atrapa explicando qué acciones se toman al respecto una vez capturadas.

- AlgoStarFinalizadoException: Esta excepción fue creada para finalizar *AlgoStar* cuando la cantidad de turnos es mayor a nueve y uno de los jugadores ya no le quedan más edificios.

- CantidadMaximaDeZanganosEnExtractorException: Esta excepción fue creada para asegurarse de que en el extractor no se exceda el límite de zánganos.

- CriaderoSinLarvasException: Esta excepción fue creada con el fin de evitar que se cree un Zángano si el criadero está vacío.

- EdificioNoEnergizadoException: Esta excepción fue creada con el fin de evitar de que se ejecuten ciertos métodos si el edificio no está energizado.

- NodoMineralYaTieneUnRecolectorDeMineralException: Esta excepción fue creada para asegurarse de que un nodo mineral no tenga

- OrigenNoEncontradoException: Esta excepción se lanza cuando el origen del moho no es encontrado.

- SinEdificoBuscadoException: Esta excepción se lanza cuando el edificio buscado no es encontrado.

- SinRecursosSuficientesException: Esta excepción fue creada para asegurarse de que un jugador no pueda ejecutar cierta acción si no tiene los recursos suficientes.

- SinUnidadBuscadaException: Esta excepción se lanza cuando la unidad buscada no es encontrada.

- UbicacionSinEdificioException: Esta excepción se lanza cuando, al querer evolucionar a un Mutalisco, la unidad/edificio no se encuentra en la ubicación deseada.

- UnidadEnConstruccionException: Esta excepción fue creada con el fin de evitar que las unidades puedan ser utilizadas mientras estén en construcción.

- ValorInvalidoDeDanioException: Esta excepción se lanza cuando el danio por recibir es menor a cero.

- ValorInvalidoParaEscudoException: Esta excepción se lanza cuando el valor del escudo establecido es menor a cero.

- ValorInvalidoParaVidaException: Esta excepción se lanza cuando el valor de la vida establecida es menor a cero.

- VolcanSinGasVespenoParaExtraerException: Esta excepción fue creada para asegurarse de que no se pueda seguir extrayendo gas si el volcán se quedó vacío.

- VolcanSinRefineriaDeGasConstruidaException: Esta excepción fue creada con el fin de evitar que se pueda extraer gas sin una refinería de gas construida sobre un volcán.

- VolcanYaTieneUnaRefineriaDeGasConstruidaException: Esta excepción fue creada con el fin de evitar que se construya otra refinería de gas sobre un volcán (propia o del enemigo).

- EdificioNoOperativoException: Esta excepción fue creada con el fin de evitar que los edificios puedan ser utilizados mientras estén no operativos.