FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — AlgoStar

[7507/9502] Algoritmos y Programación III
Curso 2
Segundo cuatrimestre de 2022

Alumno:	RUIZ DIAZ, Julián
Número de padrón:	108410
Email:	jruizd@fi.uba.ar

Alumno:	CÁCERES, Matías
Número de padrón:	101883
Email:	macaceres@fi.uba.ar

Alumno:	FERNANDEZ FOX, Joel
Número de padrón:	104424
Email:	jfernandezf@fi.uba.ar

Índice

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en desarrollar un juego basado en StarCraft, utilizando los conceptos del paradigma de la orientación a objetos, principios y patrones de diseño. Se utilizo TDD como metodología de desarrollo, además de la Integración Continua (CI), configurado en un repositorio de GitHub.

2. Supuestos

Se tomaron varios supuestos sobre el enunciado durante el desarrollo del programa. Algunos de los mas importantes fueron:

1. Una entidad no se puede construir en un área espacial, es decir, todas se construyen sobre tierra.
2. Una unidad no se puede construir sobre un recurso.
3. Una unidad requiere de su piso especifico para construirse: no se pueden construir sobre un area sin ningún tipo de piso (moho o energía de un pilón).
4. Una unidad (exceptuando el zángano) no se puede mover a un área con un recurso.
5. Una unidad solo puede atacar una vez por turno.
6. Una unidad solo puede atacar a una entidad del equipo contrario.
7. Una unidad solo puede moverse una vez por turno.
8. Una unidad solo puede moverse en un rango de 3 con respecto a la posición que ocupa.
9. Un área solo puede contener una entidad al mismo tiempo, es decir, solo una entidad puede ocupar un área.
10. Un Amo Supremo no requiere ninguna estructura para construirse.
11. Una estructura sin energía no puede proveer suministro (o población) a la raza.
12. Una entidad en construcción no puede proveer suministro (o población) a la raza.
13. Una unidad que consume suministro siempre lo hace, independientemente de si esta en construcción.
14. Una estructura sin energía no puede regenerar ni vida ni escudo.
15. Una entidad no puede construirse sobre un área con otra entidad.
16. Una unidad que revela solo lo hace luego de construirse.
17. Los criaderos son invisibles.
18. Una estructura sin energía retoma su contador de construcción al reactivarse.
19. Un scout tiene la capacidad de revelar entidades enemigas en un rango de 4.
20. Solo las unidades Zerg consumen larva para construirse, exceptuando las evoluciones del Mutalisco.

Todos los supuestos cuentan con pruebas del caso de uso.

3. Diagramas de clase

El modelo solución planteado se basa principalmente en la comunicación de Entidades, las cuales pueden ser Estructuras o Unidades. Todas ellas se componen de un Escudo y una Vida; de un estado de invisibilidad, que determina si pueden ser atacadas o no; de un estado operativo, que determina si pueden atacar, ejecutar acciones, como pasan el turno, y si aportan o no suministro a la raza; y de una clase AfectaSuministro, que determina la variación puntual del suministro. Todas tienen un área que ocupan y la raza a la que pertenecen.

Cada clase concreta conoce a las clases Construibles, que son las encargadas de determinar si se pueden construir dadas las condiciones pasadas por parámetro.

Los estados conocen a los comandos, que, dependiendo de cual estado este presente en la entidad, se ejecutan o no.

Se omiten en este informe los diagramas que muestran la herencia de las distintas clases concretas.

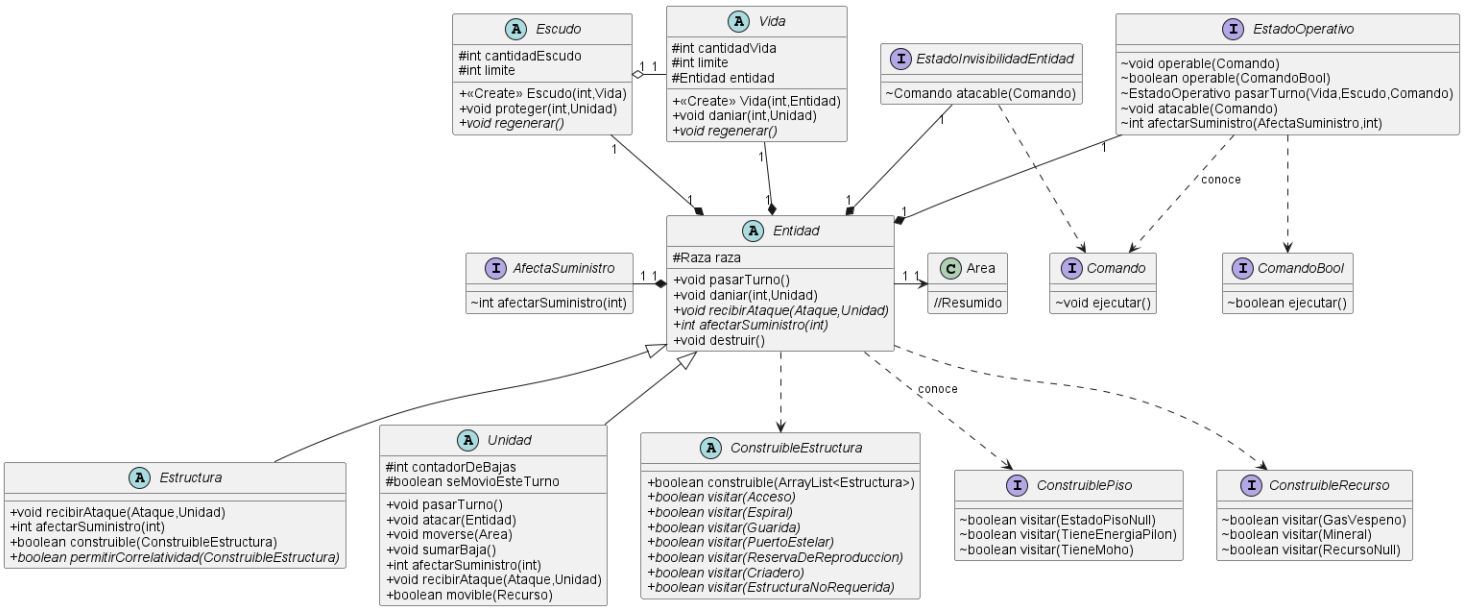

Figura 1: Diagrama de clases de una Entidad.

Luego, las Unidades cuentan con dos clases mas que las componen. El tipo de unidad determina el movimiento que pueden tener por las áreas y si pueden ser atacadas o no, y el ataque determina si atacan o no, además de cuanto daño (aire o tierra) hacen y en que rango.

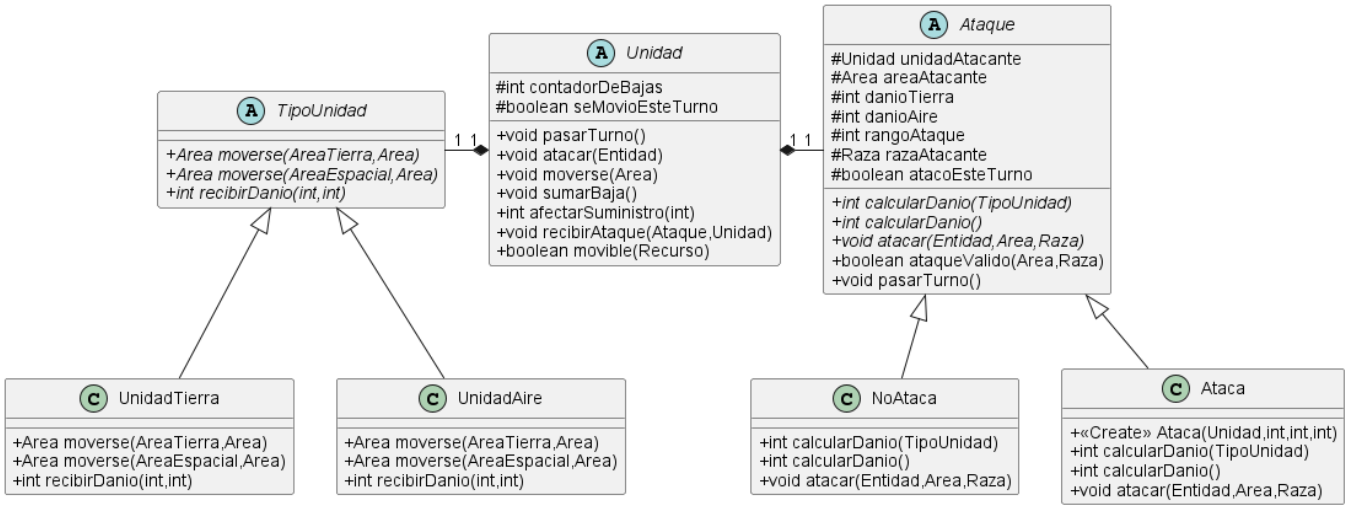

Figura 2: Diagrama de clases de una Unidad, obviando la herencia de las subclases.

Las áreas son la otra mitad del modelo solución, y representan las celdas del tablero o mapa del juego. Funcionan como una suerte de fachada, agrupando el comportamiento de otras clases. Se componen de una coordenada, encargada de manejar las relaciones numéricas entre distintas áreas; de un estado de ocupación, que determina si una unidad se puede mover o si se puede construir sobre ella; de un estado del piso, que se actualiza en base a los pilones y el moho en el mapa, permitiendo la construcción para las distintas razas; de un recurso, del cual algunas entidades pueden extraer; y de un tipo de área, que junto con los tipos de unidades, determinan el movimiento y la construcción posible.

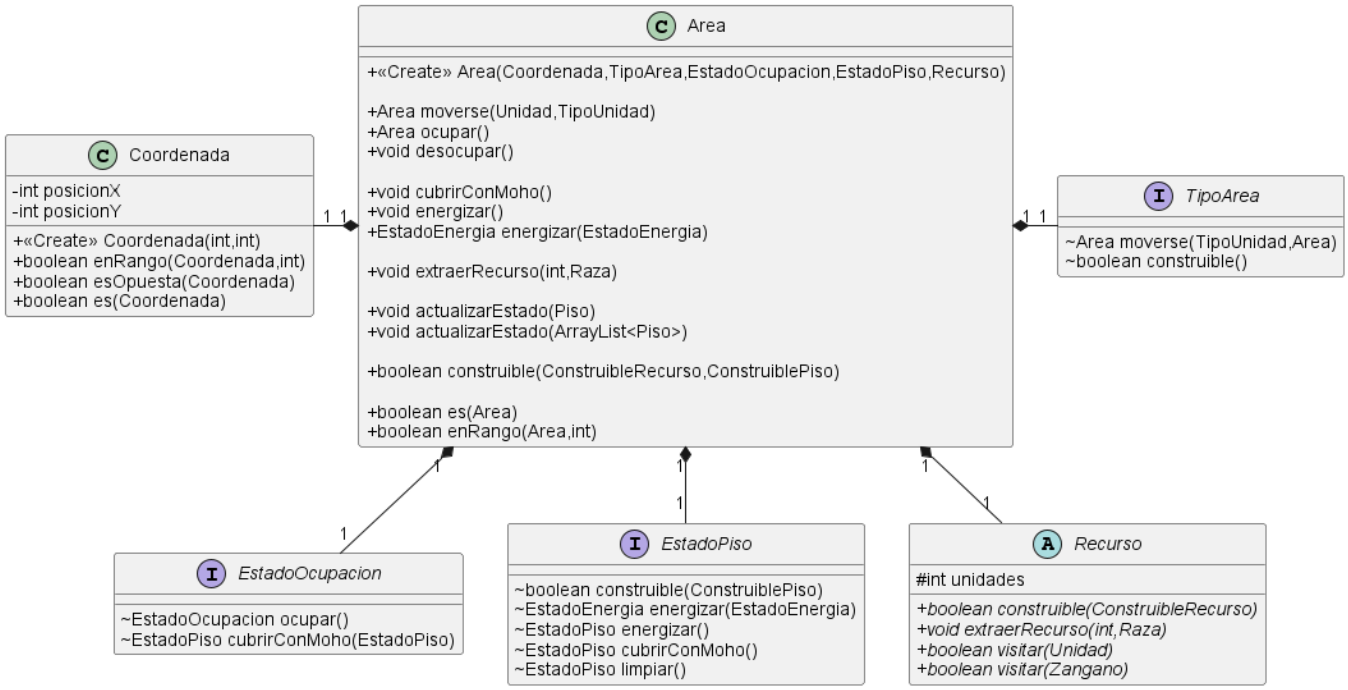

Figura 3: Diagrama de clases de un Área.

Las razas son las encargadas de guardar y permitir la construcción de las distintas entidades,

utilizando constructores, en base a los recursos que guarden sus reservas. Al tener todas las entidades de un jugador, son las encargadas de iterar sobre todas ellas (al pasar turno o calcular el suministro total, por ejemplo). Además, conocen a la raza contrincante, permitiendo revelar entidades enemigas o revelar una unidad propia que se movió en el mapa. Finalmente, determinan si un juego se termina, en base a si tienen estructuras o no. Los Protoss y los Zerg son clases concretas que heredan de raza, y estas contienen solamente los constructores de las entidades que les corresponde a cada una, además de otros métodos que solo tengan sentido ejecutarse en una u otra.

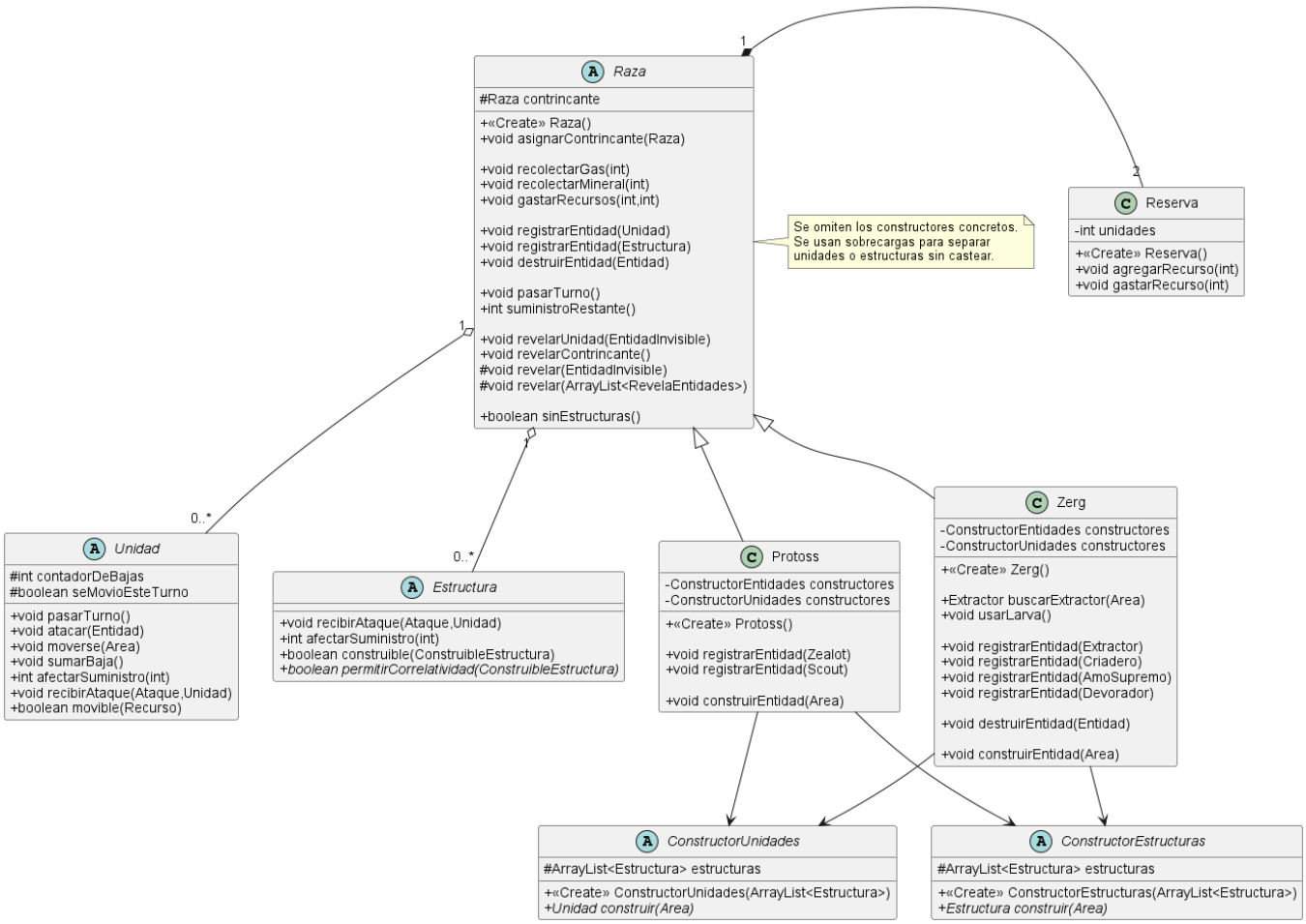

Figura 4: Diagrama de clases de una Raza.

Finalmente, el Juego es el encargado de guardar ambas razas, registrar los jugadores en base a sus nombres y colores, pasar el turno a todas las entidades y pisos presentes en la partida, además de determinar cual de las dos razas juega, y si alguno perdió. Tiene una referencia al Mapa, el cual se planteo como Singleton.

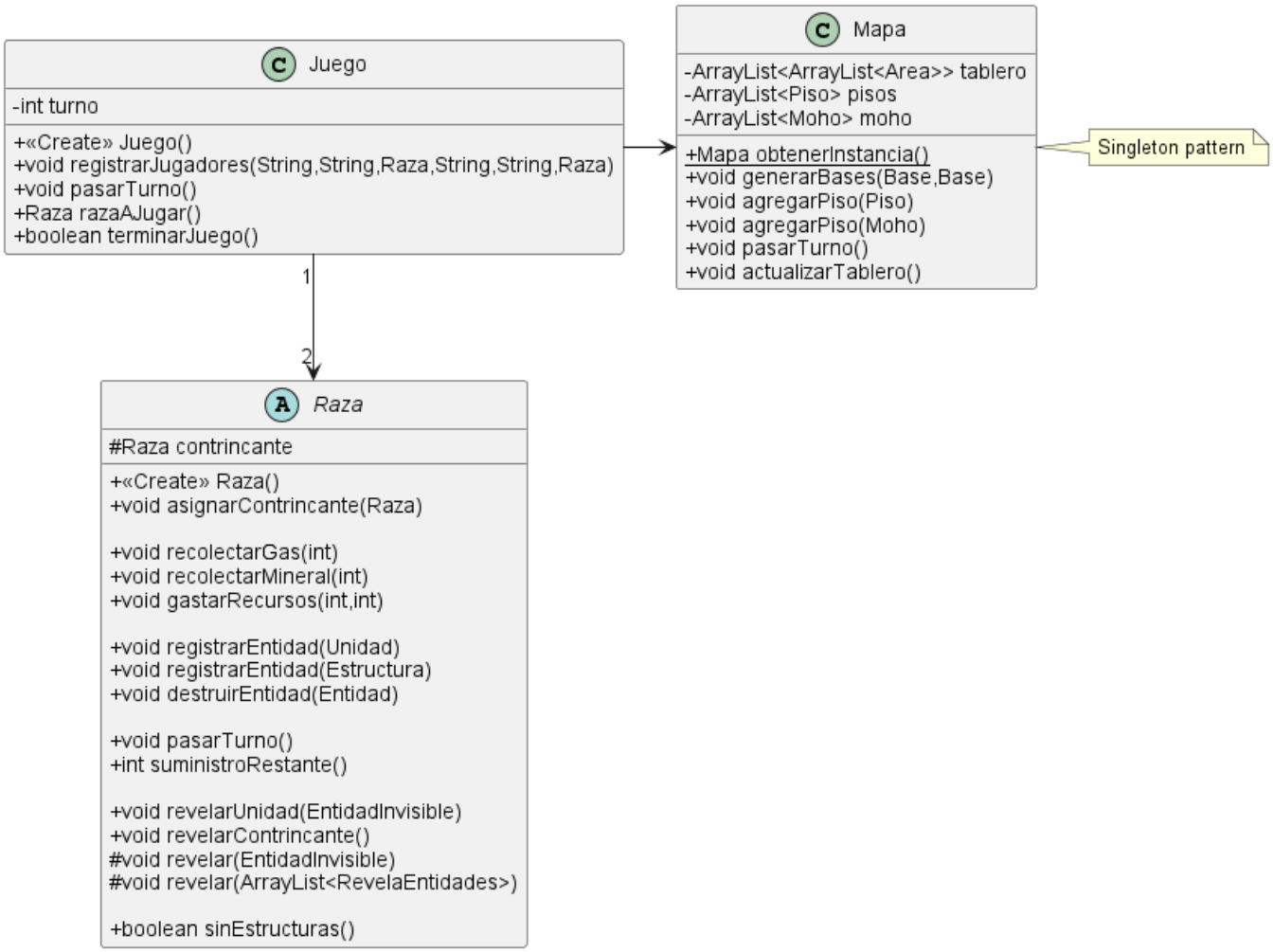

Figura 5: Diagrama de clases del Juego.

Todos los diagramas de clase anteriores, así como otros de clases mas especificas, pueden encontrarse en la carpeta de diagramas en el repositorio.

4. Diagramas de paquetes

Por el tamaño de la imagen, el diagrama de paquetes se puede ver en la carpeta de diagramas en el repositorio.

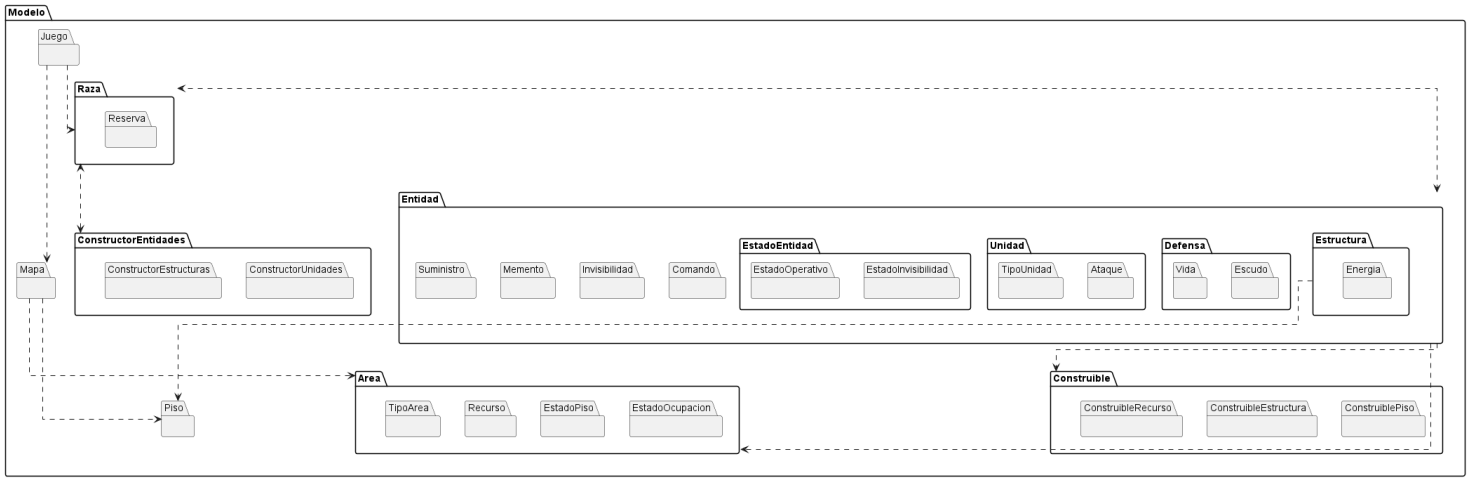

Figura 6: Diagrama de paquetes del modelo.

5. Diagramas de secuencia

A continuación se muestran diagramas de las secuencias mas relevantes del trabajo:

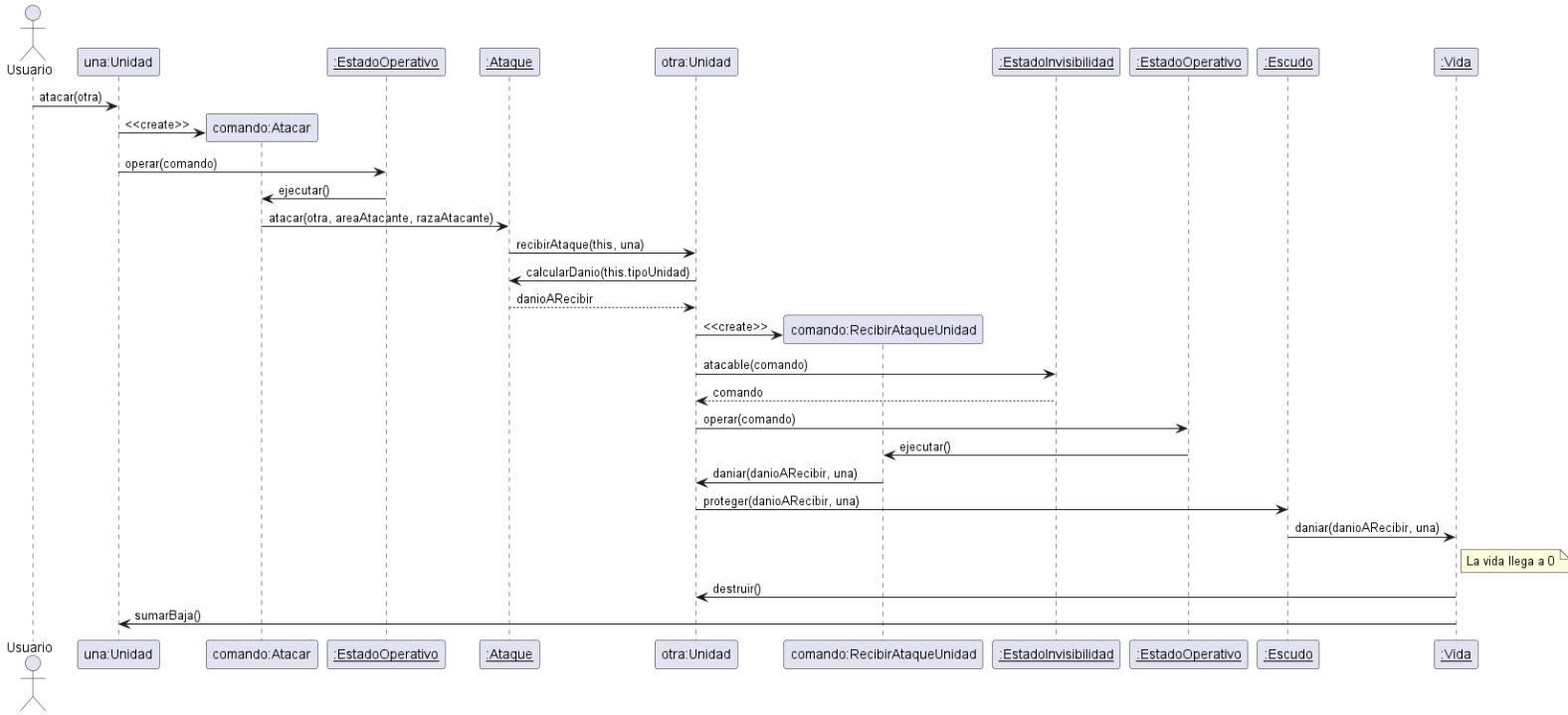

Figura 7: Diagrama de secuencia del ataque de una unidad a otra, y su destrucción.

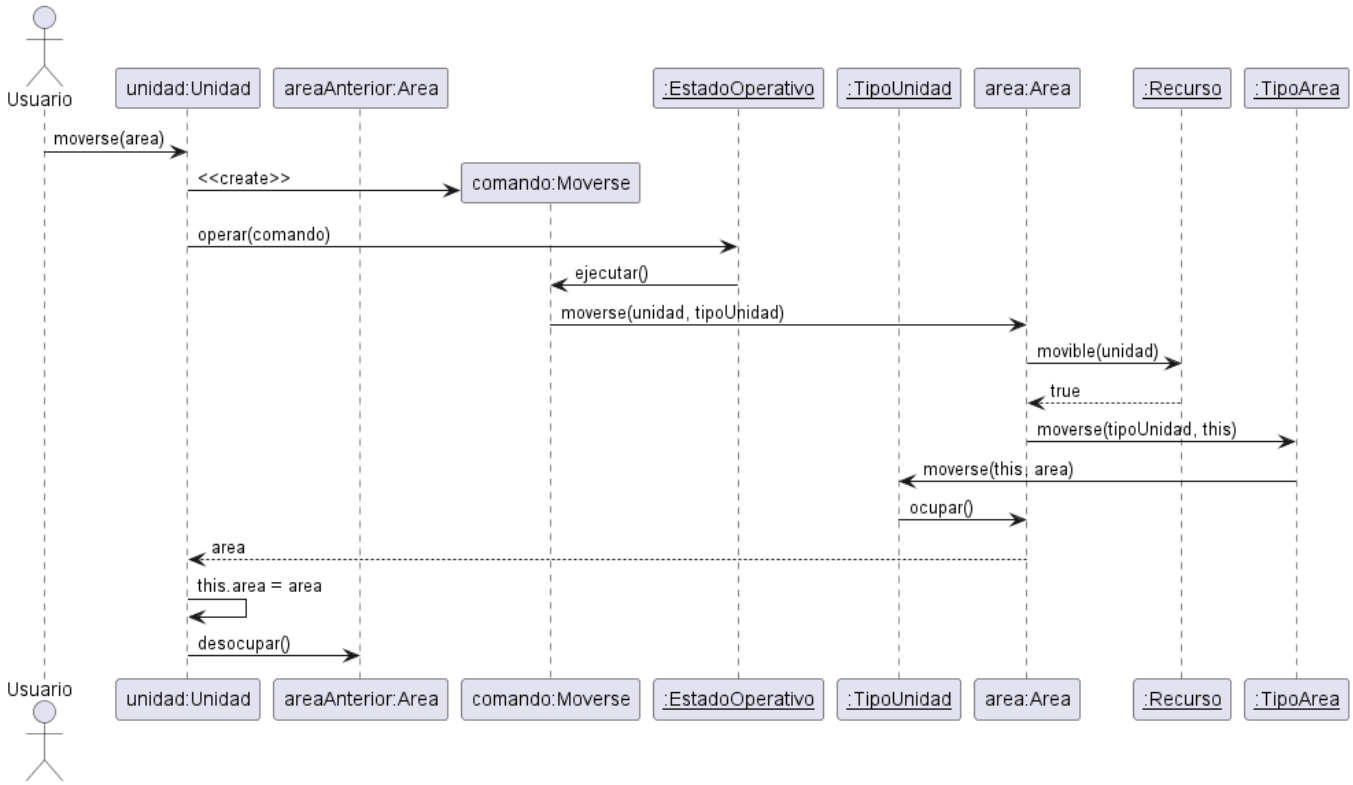

Figura 8: Diagrama de secuencia del movimiento de una unidad a un área.

6. Diagramas de estados

A continuación se muestran diagramas de estado de las clases mas importantes del trabajo. Se omiten estados de clases como la Invisibilidad y la Energía.

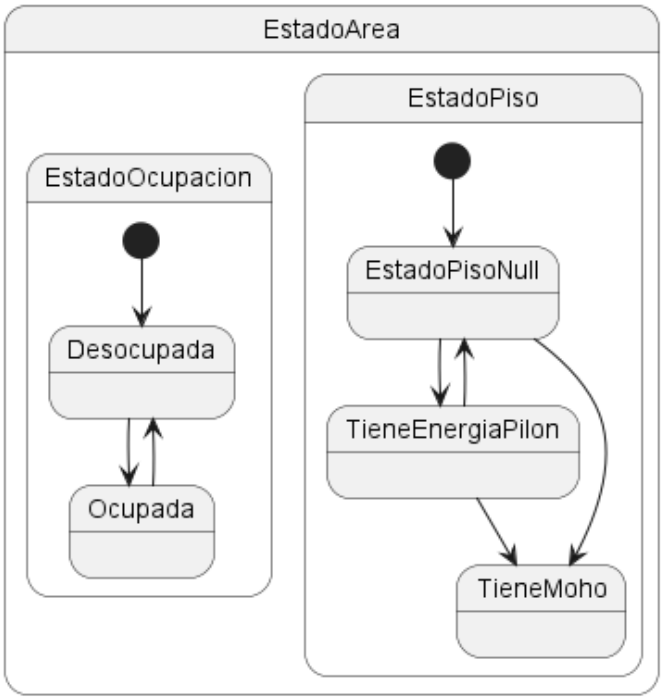

Figura 9: Diagrama de estado de un Área.

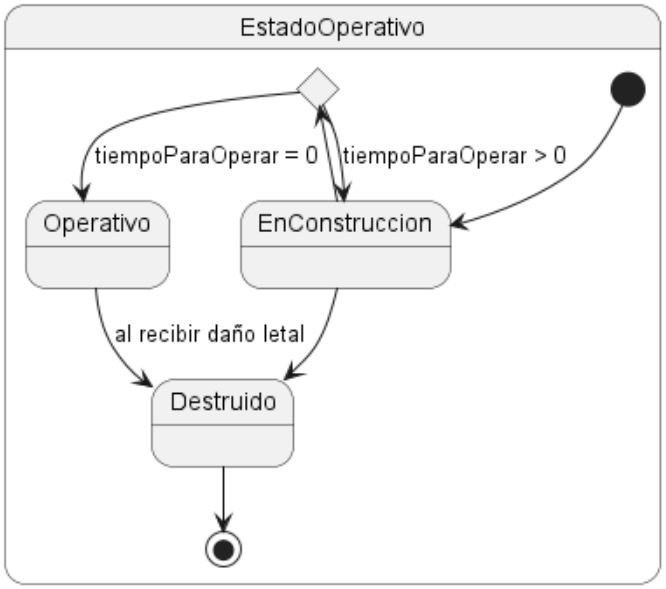

Figura 10: Diagrama de estado Operativo para entidades que no utilizan energía.

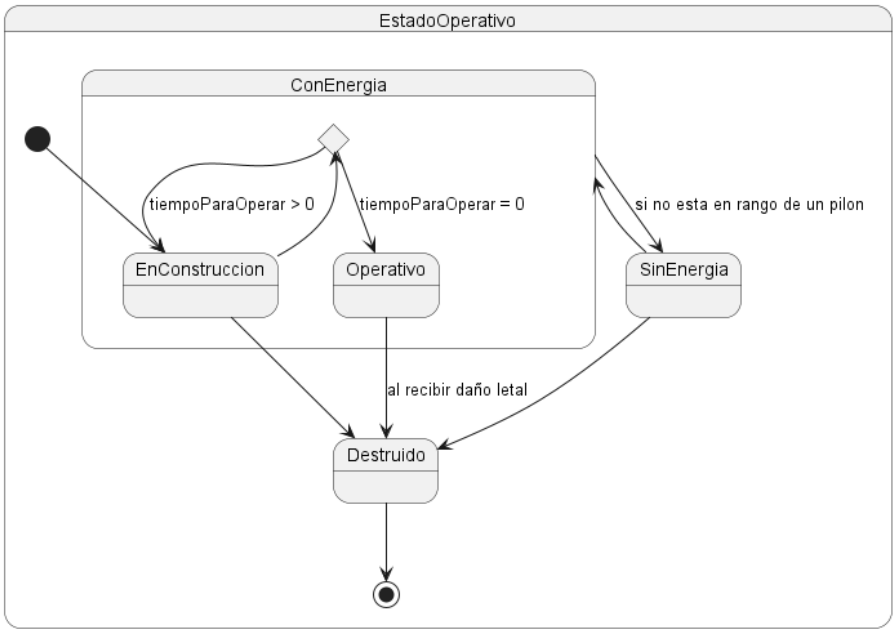

Figura 11: Diagrama de estado Operativo para entidades que utilizan energía.

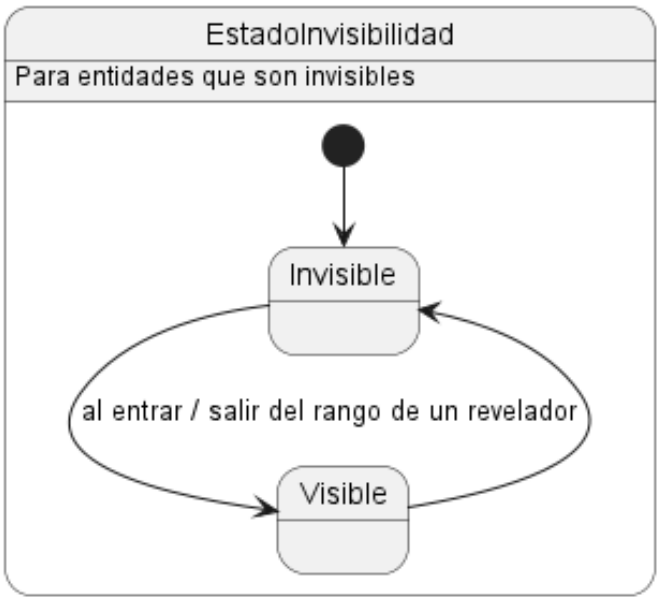

Figura 12: Diagrama de estado de la Invisibilidad de algunas entidades.

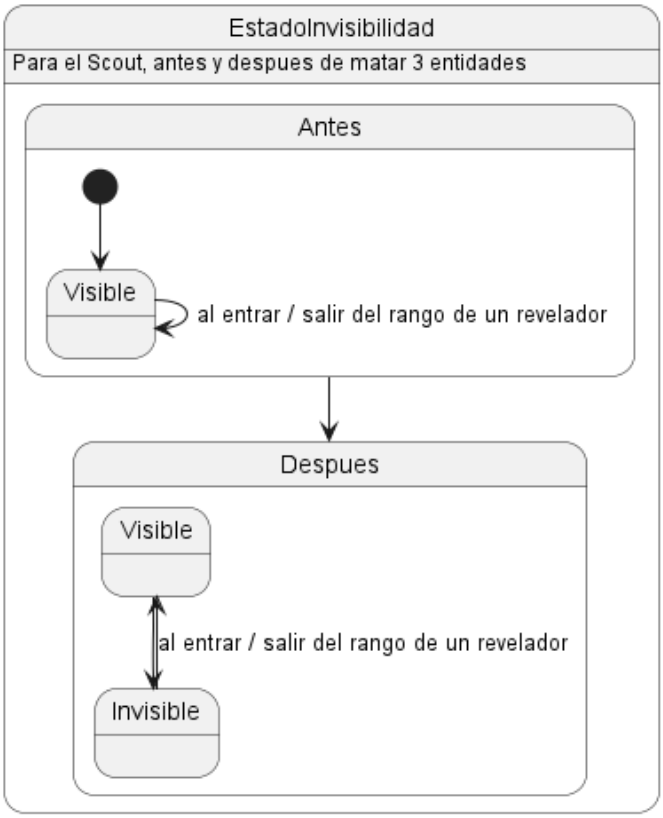

Figura 13: Diagrama de estado de la Invisibilidad para un Scout.

7. Detalles de implementación

7.1. Principios de diseño

Para resolver este trabajo practico aplicamos la metodología TDD, escribiendo el mínimo código indispensable para cumplir incrementalmente con los distintos casos de uso planteados. Se intentaron plantear desde un principio cuales serian las interfaces que deberían existir para no depender de implementaciones especificas, cumpliendo asi con uno de los principios SOLID (segregación de interfaz). También, se intento crear nuevas clases por cada nuevo comportamiento introducido en el modelo, cumpliendo con el principio de responsabilidad única, aunque luego resulto en muchas clases creadas, y en algunos casos, implico sobrecomplejizar el modelo. Se intento cumplir con el principio de inversión de dependencia, utilizando interfaces o clases abstractas dentro de los métodos y atributos de las distintas clases. De esta forma, se decidió usar herencia en los casos de las Estructuras y Unidades, siendo todas Entidades, y luego cada una de las clases concretas que heredan tanto de Estructura o Unidad, agrupando solamente los comportamientos similares o los mensajes polimórficos necesarios (como pasar turno o dañar). Todos los demás comportamientos se delegaron a otras clases. Por ejemplo, un Área delega lo relacionado a su coordenada a la clase Coordenada, o las Unidades delegan su ataque a la clase Ataque.

7.2. Patrones de diseño

Se utilizaron varios patrones de diseño para resolver distintos problemas. Particularmente:

1. Patrón Strategy: se utilizo, por ejemplo, en el ataque de las unidades. Se decidió por este patrón porque una Unidad no cambia su estrategia de ataque durante el juego. Por tanto,

los distintos ataques no se conocen entre si. Con este patrón solucionamos el problema de que unas clases implementaban el ataque mientras que otras no.

2. Patrón State: fue el patrón mas utilizado en este modelo solución. El uso mas importante fue el del EstadoOperativo de las entidades, que influye en muchos comportamientos posibles o no dentro del juego. Se decidió por un state en este y otros casos porque se necesitaba la actualización constante del estado de distintas clases, para determinar, por ejemplo, si una unidad puede atacar, si una estructura puede permitir una construcción, entre otras. De esta forma, las distintas clases creadas son las que determinan si el estado del invocante debe cambiar o no, devolviendo la instancia correspondiente, sin que otros se enteren o puedan cambiarlo.

3. Patrón Memento: se utilizo para restaurar correctamente el estado anterior de una estructura Protoss que se queda sin energía. De esta forma, implementamos uno de los supuestos anteriormente mencionado. También se utilizo para restaurar correctamente la invisibilidad de una entidad que fue revelada.

4. Patrón Command: se utilizo para mandar comandos al EstadoOperativo de una entidad, el cual decide si se ejecutaba o no. De esta forma, solucionamos el problema de que este estado es el responsable de permitir o no muchas acciones, sin la necesidad de tener un método especifico para cada uno de los casos, logrando así desacoplar las distintas clases.

5. Patrón Factory Method: se utilizo para construir todas las distintas unidades y estructuras, ya que se contaba con una interfaz común. De esta forma, pudimos resolver el problema de que una raza pueda crear entidades que no les corresponde, además de simplificar y estandarizar el llamado de creación de una entidad (solamente requiere el área a construir).

6. Patrón Singleton: se utilizo para contar son solamente una instancia del mapa, logrando así que se generen distintos mapas, y consecuentemente, distintas áreas.

7. Patrón Facade: se intento aplicar el patrón Facade para agrupar en una sola clase el comportamiento de otras clases interdependientes. Concretamente, el Área conforma en mayor o menor medida, una fachada, asegurando así la coherencia de los distintos estados.

8. Patrón Visitor: se intento aplicar este patrón para desacoplar algunos comportamientos que dependen de dos clases concretas (double-dispatch). En nuestro modelo, los Construibles apuntan a resolver el problema de las estructuras que habilitan o no la construcción (correlatividades).

8. Excepciones

Exception AtaqueNoValido: excepción genérica lanzada cuando una unidad no puede atacar a otra entidad, ya sea porque esta fuera de rango, no tiene el tipo de daño correcto, o porque esta atacando a una entidad propia.

Excepcion BaseNoOpuesta: excepción creada con el objetivo de atrapar el caso en que una base no se genera de forma opuesta a la otra, asegurando un comienzo de partida justo.

Excepcion ConstruccionNoValida: excepción genérica lanzada cuando una entidad no se construye en el contexto correcto, ya sea porque el piso no tiene moho o energía de un pilón, no se cuentan con los recursos suficientes, etc. Se atrapa para evitar directamente que un jugador construya esa entidad desde el constructor en la raza.

Excepcion CriaderoSinLarvas: excepción lanzada cuando un criadero no tiene larvas para engendrar otra entidad. Se atrapa para revisar si los Zerg tienen alguna larva en alguno de sus Criaderos.

Excepcion EntidadDestruida: excepción que se lanza cuando se intenta operar o atacar una entidad destruida, y se atrapa para evitar la acción.

Excepcion EntidadNoOperativa: similarmente, se lanza esta excepción cuando una estructura quiere operar, o cuando una unidad se intenta mover o atacar cuando aun no puede (por ejemplo, cuando esta en construccion).

Excepcion EvolucionNoValida: excepción lanzada cuando un Mutalisco intenta evolucionar en condiciones no validas. Se atrapa para evitar la acción, evitando gastar recursos y larvas.

Excepcion ExtractorLlenoException: excepción lanzada cuando un Zangano intenta moverse a un área con un Extractor pero el extractor esta lleno. Se atrapa para evitar la acción.

Excepcion JugadoresNoCompatibles: se lanza cuando los jugadores eligen la misma raza, evitando el comienzo de la partida.

Excepcion MovimientoNoValido: se lanza cuando una unidad desea moverse a un área incorrecta (por ejemplo, una unidad de tierra moviéndose a un área espacial) o cuando intenta moverse a una posición ocupada. Se atrapa para evitar la acción.

Excepcion MovimientoSobreRecurso: se lanza cuando una unidad se mueve a un área con un recurso. En todos los casos, se evita la acción, con la excepción del Zangano, el cual atrapa la excepción para probar si es un extractor y así concretar el movimiento.

Excepcion NombreNoValido: se lanza cuando el nombre es muy corto, y evita el inicio de la partida.

Excepcion PosicionOcupada: se lanza cuando una unidad intenta moverse a un área que ya esta ocupada, o cuando una entidad se quiere construir sobre ella. Se atrapa para evitar la construcción o el movimiento.

Excepcion RazaZergSinLarvas: se lanza cuando los Zerg no tienen larvas. Atrapa la excepción de CriaderoSinLarvas para hacer el chequeo, y se atrapa para evitar la construcción de una unidad Zerg.

Excepcion RecursoInsuficiente: se lanza cuando una reserva no tiene los recursos suficientes para construir una entidad. Se atrapa en la misma secuencia y se arroja la excepción ConstruccionNoValida, la cual es atrapada para evitar la acción desde la raza.

Excepcion RecursoVacio: se lanza cuando un recurso ya fue extraído totalmente.

Excepcion SuministroInsuficiente: se lanza cuando una unidad se quiere construir y la raza no cuenta con el suministro suficiente, y se atrapa para evitar la acción.

Excepcion UnidadYaAtaco: se lanza cuando una unidad quiere atacar mas de una vez en un mismo turno, y se atrapa para evitar el ataque.

Todas las excepciones fueron creadas con el objetivo de ser atrapables posteriormente desde el controlador, evitando directamente la ejecución de una acción no valida en el contexto de la partida.