Facultad de Ingeniería

Universidad de Buenos Aires

Departamento de Computación

Algoritmos y Programación III

2^{do} cuatrimestre 2019

Trabajo Práctico 2:
AlgoChess

Autores:

Joaquín Elordi (96229)
gelosofederico@gmail.com

Federico Geloso (98138)
gelosofederico@gmail.com

Cristina Kuo (97777)
cristinaa.kuo@gmail.com

Cristian Torales (95549)
crist.torales@gmail.com

18 de diciembre de 2019

Índice

1. Objetivo

Desarrollar una aplicación utilizando un lenguaje de tipado estático (Java), con un modelo orientado a objetos y trabajando con las técnicas de TDD e Integración Continua.

2. Supuestos

- Cuando el jugador eligió las unidades con su contador de puntos y le sobran puntos pero estos no alcanzan para elegir otra unidad, no se ofrece la opcióe seguir eligiendo unidades.
- En el inicio del juego, se decidió que se turnen los jugadores para la colocación de las unidades en el tablero, esto es para no dar ventajas tácticas al segundo jugador ya que conoce la disposición final de las unidades del jugador contrario.
- Se supone que en cada turno se pueden elegir las opciones de atacar o de moverse.
- Se esperan dos jugadores para que el juego se ejecute.
- El jugador puede mover a las unidades en cualquiera de las 8 direcciones: este, oeste, norte, sur, noreste, noroeste, sureste y suroeste.
- Cada batallón está formado por 3 soldados y se genera a partir del movimiento de un soldado. Cuando se mueve un soldado, se verifica si se forma o no el batallón, en caso de que se forme el mismo, se moverán en conjunto. En caso contrario, solo se moverá el soldado que fue indicado para que se mueva.
- No se ofrece la opción de que un soldado dentro de un batallón se pueda mover solo.

3. Desarrollo

El desarrollo de la aplicación fue incremental, realizando entregas intermedias donde se agregaron nuevas fucionalidades al modelo. Se utilizaron otras metodologías del desarrollo de software asociadas a *"Extreme Programming"*. Entre ellas la integración continua para comprobar que la calidad del código se mantenga alta, corriendo los tests en cada push. Esto requiere que estas pruebas sean lo más exhaustivas posibles para que se pueda asegurar que el código esté funcionando correctamente cada vez que estas pruebas son corridas. Para esto, se utilizó la metodología de TDD, por lo que la mayor parte del código del modelo está testeado.

El programa está organizado según el patrón MVC, por lo que los principales paquetes en el código útil son modelo, controlador y vista. El modelo tiene su paquete de pruebas unitarias en la carpeta de tests.

Para lograr cumplir con estándares de calidad de software y perpetuar buenas prácticas de programación orientada a objetos se siguieron como guia a la hora de diseñar las clases, metodos y las interacciones entre los objetos que intervienen en un proceso los principios de diseño SOLID. Además de algunos de los patrones de diseño conocidos para resolver determinados problemas.

4. Diagramas de clase

Las primeras clases con de mayor jerarquía son Unidad, Tablero y Jugador, dado que son inherentes al dominio del problema. Unidad representa a las unidades que se pueden poner,

por lo que una de las primeras abstracciones fue que esto fuera una clase de la que heredan los distintos tipos de unidades: Soldado, Jinete, Catapulta y Curandero. Estos responden a los mismos mensajes de atacar, moverse, ser colocados y recibir daño, pero algunos de estos métodos fueron implementados en la clase Unidad para evitar la repetición de código, dado que el comportamiento según algunos de estos mensajes era idéntico, mientras que otros no.

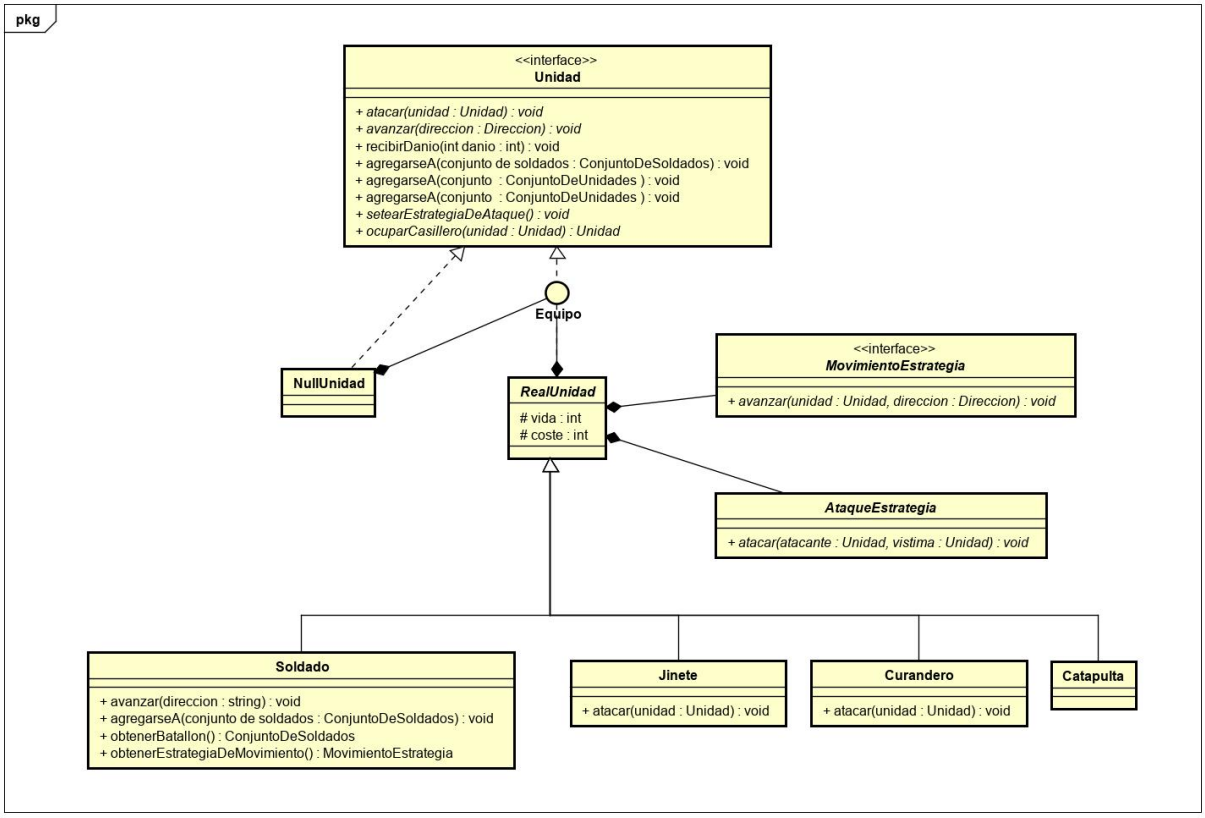

Figura 1. Diagrama de clases de Unidad.

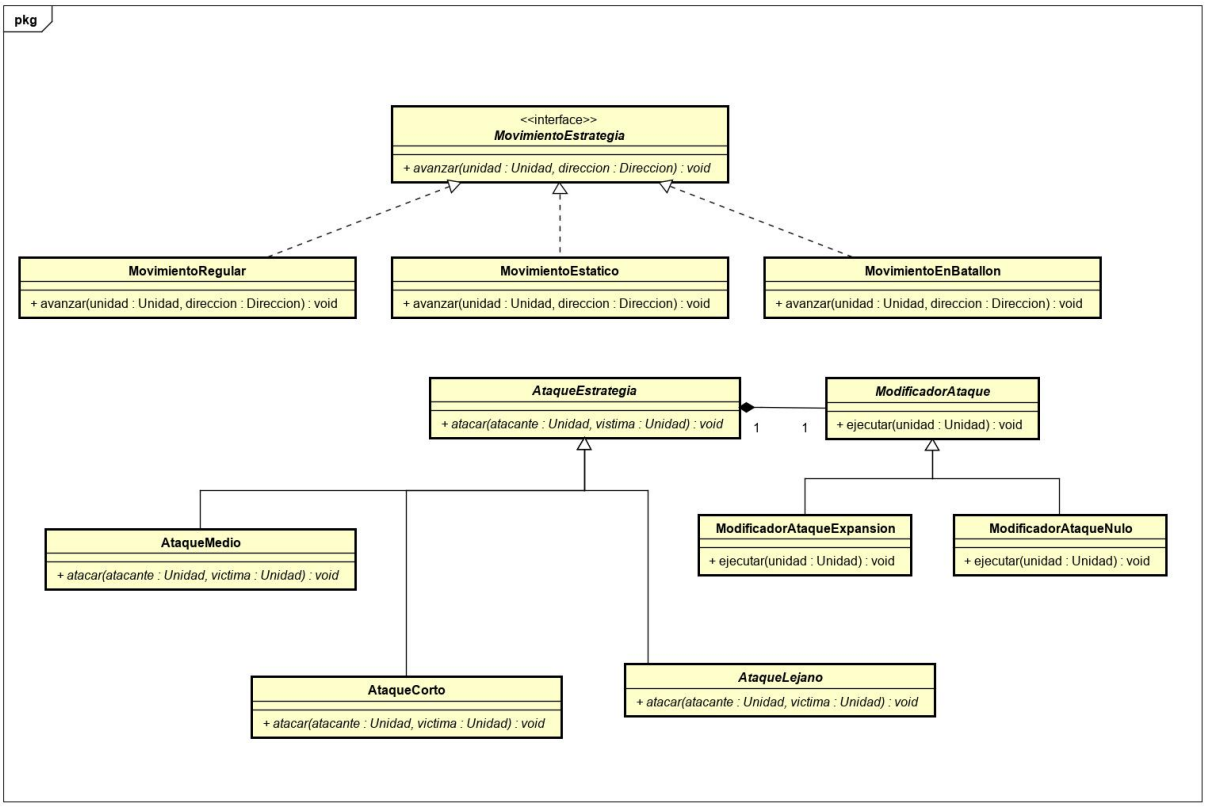

Figura 2. Diagrama de clases de las estrategias de movimiento y de ataque.

En la figura 3 se puede ver como la entidad Unidad fue creciendo en abstracción al agregarse nuevas funcionalidades para el modelo. En principio Unidad se transformó en una intefaz donde se definen las forimas de los métodos que definen el comportamiento de la entidad unidad. Luego, esta interfaz es imlementada por la clase abstracta RealUnidad donde son implementados lla mayoria de los metodos comunes a todas las clases hijas para evitar la replicacion de código. A su vez la clase ReaUnidad implementa la interfaz Movimiento-Estrategia Y la clase abstracta Ataque estrategia a las cuales le delega el comportamiento de los metodos atacar y avanzar utilizando el patron de diseño *Strategy.*

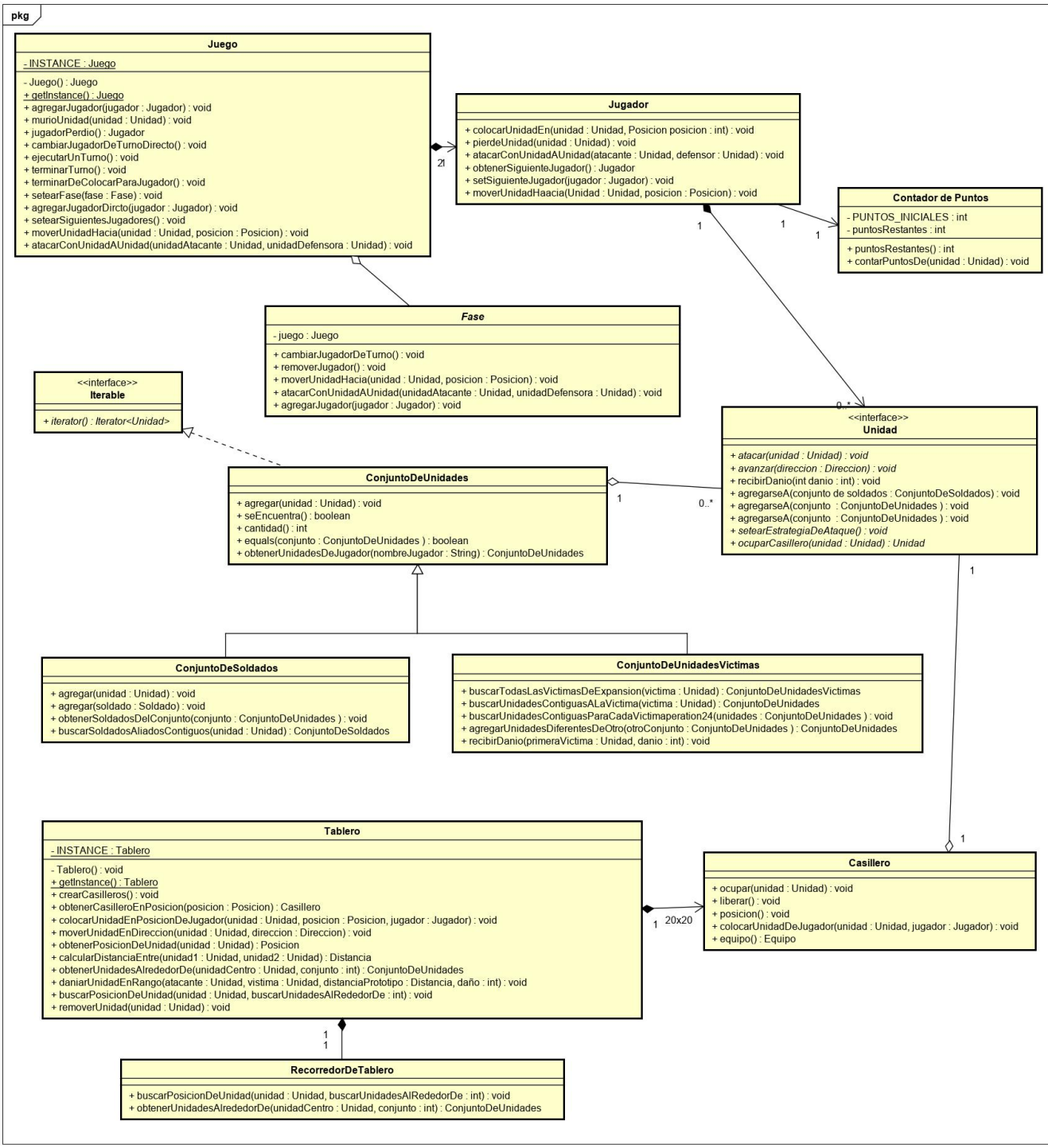

Figura 3. Diagrama de clases de Juego.

La clase Juego es la clase con mayor nivel de abstración, delega tareas a las demás entidades del modelo y se encarga de inicializar y moderar la inicialización y el desarrollo del juego.

La clase Tablero está inherentemente involucrada en la mayoria de los procesos llevados a cabo para realizar las diferentes funcionalidades requeridas por el dominio de la aplicación. Para intentar evitar un alto acoplamiento entre clases y paquetes del modelo se decidió se propuso implementar el patron *Singleton* para resolver este problema, ponderando la alta cohesion y el bajo acoplamiento sobre el principio de responsabilidad única. Se considera una de las soluciones de diseño llevadas acabo durante el diseño del modelo.

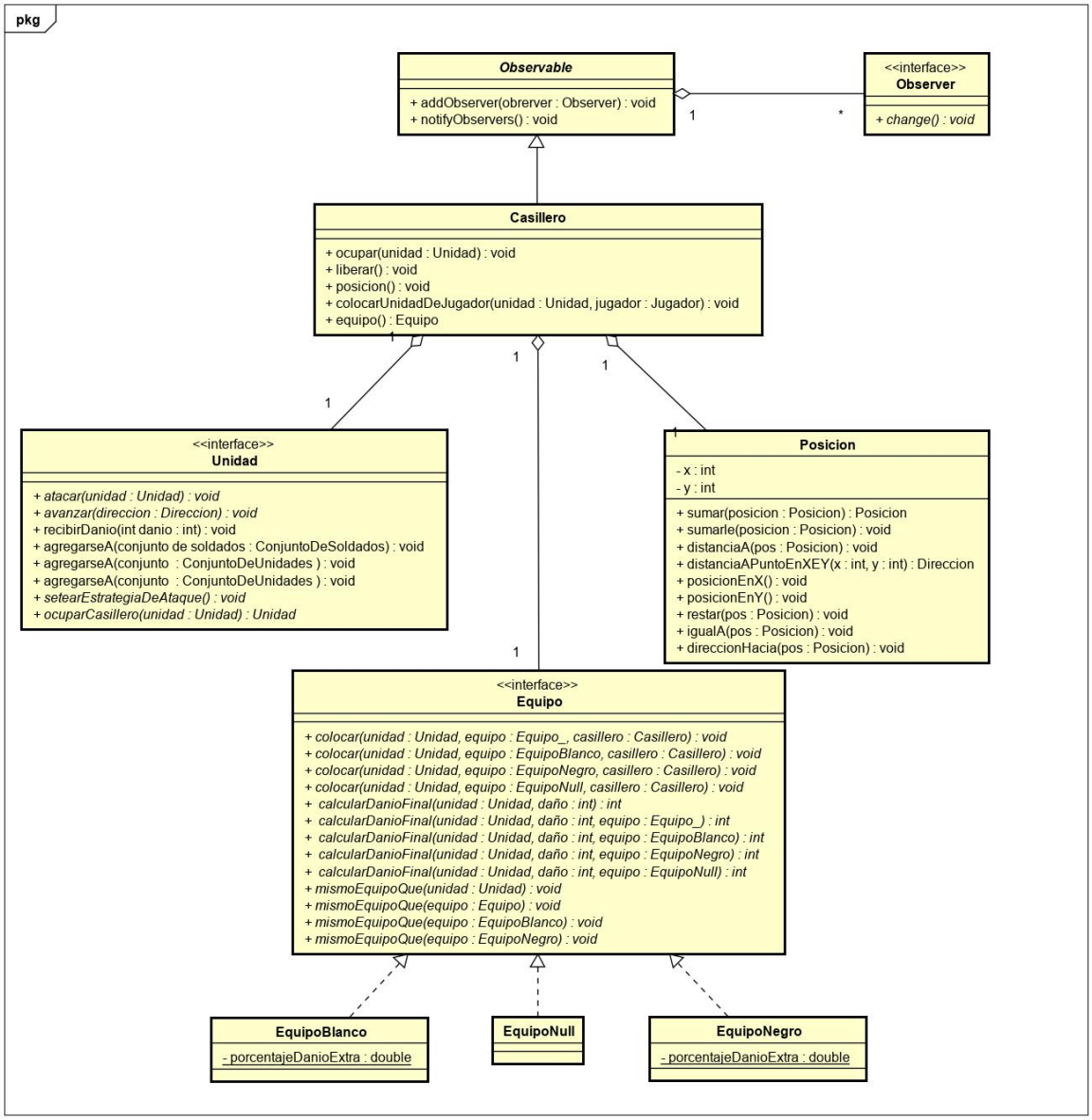

Figura 4. Diagrama de clases de Casillero.

La clase Casillero esta modelada como la celda del tablero, esta contiene una posicion, una unidad y está compuesta por uno de los dos equipos disponibles en el juego.

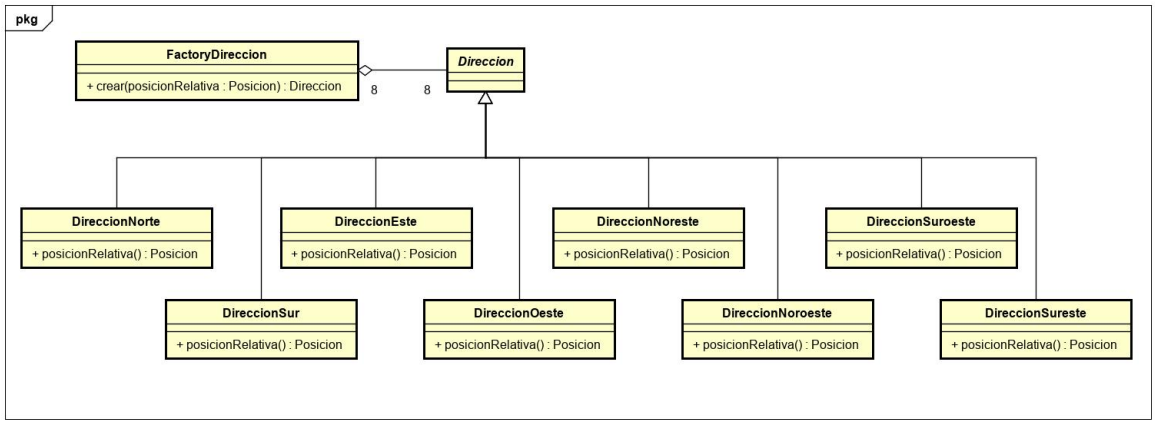

Figura 5. Diagrama de clases de Dirección.

La clase Direción fue creada con el objetivo de modelar la manera de mover unidades dentro del tablero para obtener posiciones dentro del tablero.

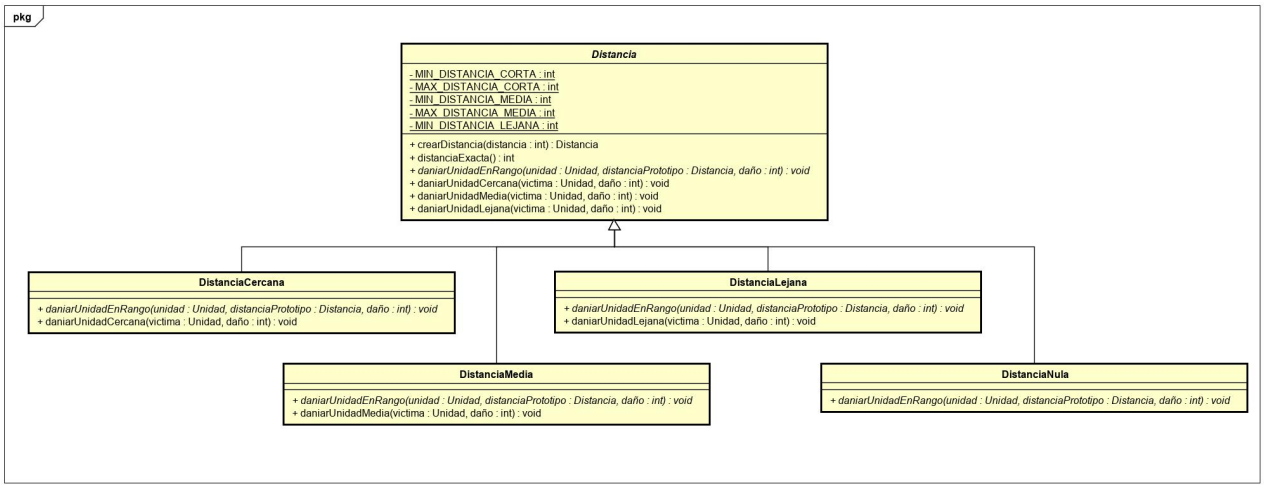

Figura 6. Diagrama de clases de Distancia.

La clase disteancia fue implementada con el propósito deresolver cuentiones de validaciones de distancias permitidas para realizar ataques entre unidades, utiliza el patron *Double Dispatch* en sus métodos.

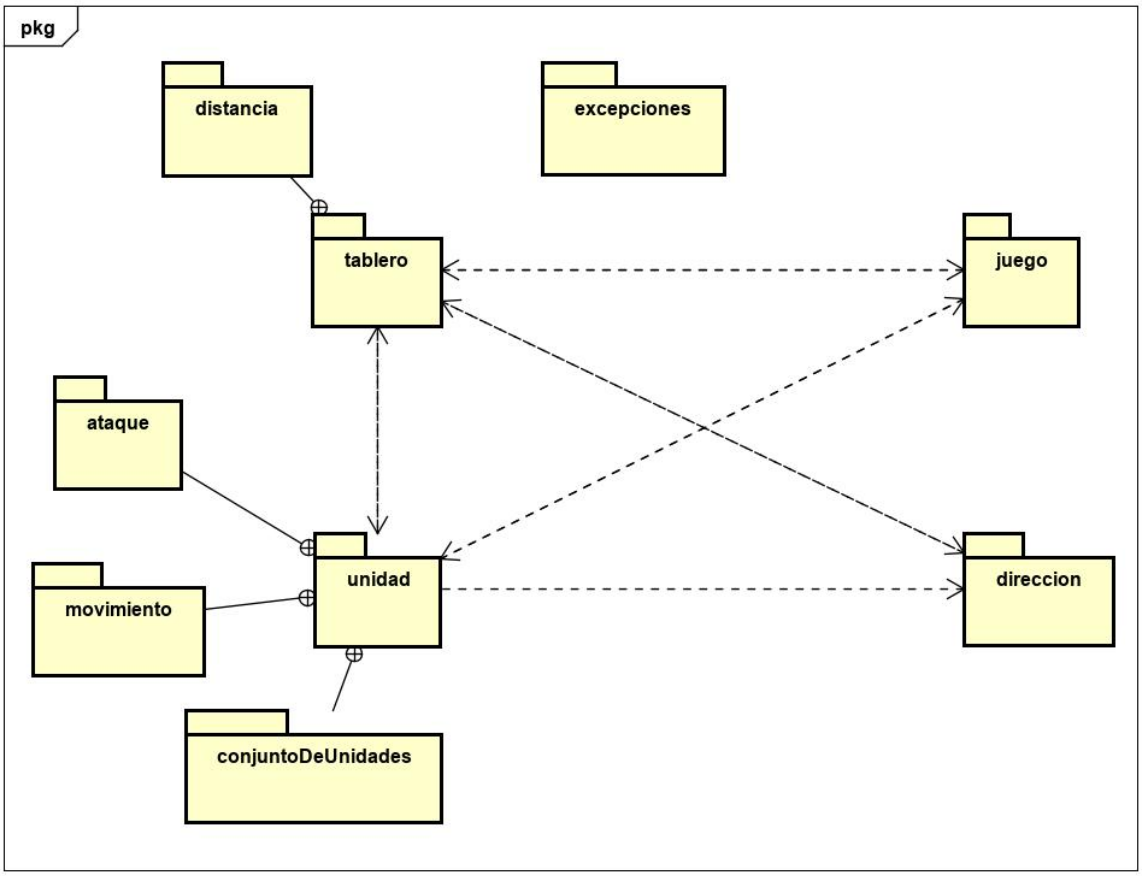

Figura 7. Diagrama de Paquetes dentro del main.modelo

5. Diagrama de estados

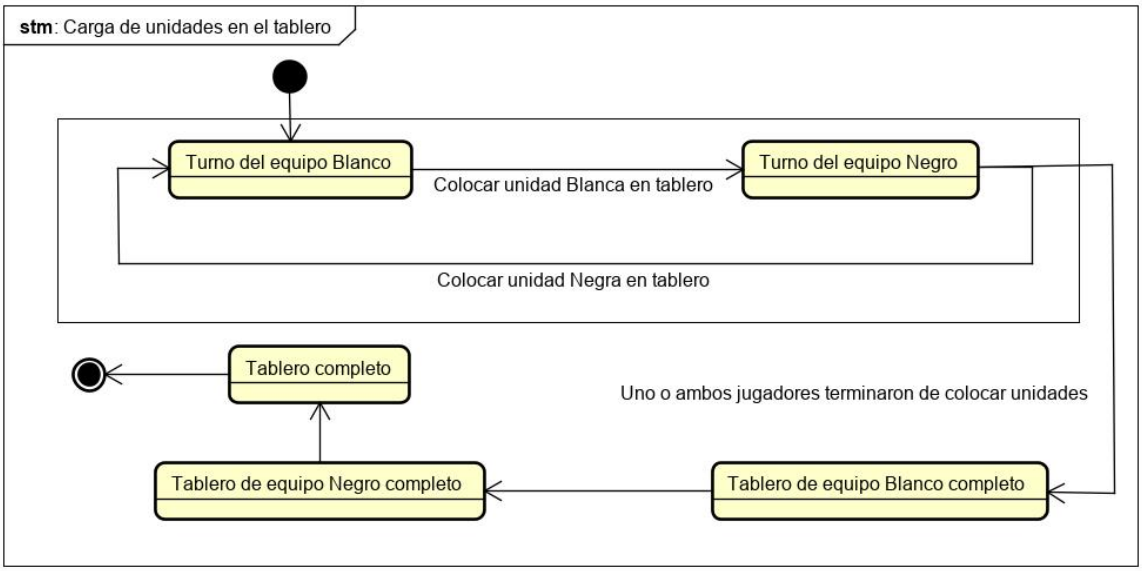

Figura 8. Diagrama de estados correspondiente a la carga de unidades en el tablero.

6. Detalles de implementación

En los requerimientos se pide que el Jinete tenga distintos ataques permitidos según su entorno, y que el soldado se mueva distinto también según su entorno, lo cual indica que la estrategia de ataque y de movimiento deben ser intercambiables en tiempo de ejecución. Por esta razón, se englobó el comportamiento del ataque y el movimiento en una clase a la que se delega, y esta es cambiada por la unidad en los casos mencionados. Esto sigue el patrón *Strategy*, y son la interfaz MovimientoEstrategia y la clase abstracta AtaqueEstrategia. Con esto también se puede evitar la repetición de código reutilizando las clases para los ataques o movimientos que son similares.

En el caso del ataque, se separó en clases de ataque según las distancias, las cuales se inicializan con el daño que hacen. Además se extendió el ataque lejano haciendo una clase que hereda de este para crear el ataque de catapulta, por lo que se extendió la funcionalidad si tocar la anterior.

Para la obtención de la distancia, se utilizó un *double dispatch* entre las clases de Distancia. Esto tiene algunas desventajas, ya que una nueva distancia requiere modificar el código de la clase madre para agregar el mensaje que pueda ser entendido por todas las otras. Sin embargo, fue la mejor solución pensada ya que se hace uso del polimorfismo para manejar los diferentes casos, lo cual permite extender el código relativamente fácil. Entonces, si bien Distancia es abierta a modificaciones debido al *double dispatch*, también es relativamente abierta a la extensión.

Otro detalle importante en las unidades es que se utilizó el patrń *NullObject* para representar los casilleros que no tienen unidades, y poder utilziar el mensaje colocar() indistintamente, sin la necesidad de preguntar si la unidad del casillero es *null* para intentar colocar una unidad real en él.

Se ha utilizado la clase *ConjuntoDeUnidades* tanto para la creación del batallón de soldados como para la creación del conjunto de víctimas de los ataques de la catapulta.

Ante la necesidad de realizar inicializaciones de clases con *null*, se utilizó el patrón de diseño *null object* que consiste en definir clases con comportamientos nulos y así poder reemplazar la utilización de *null*. Por ejemplo, esto se implementó para Unidad, la clase es NullUnidad.

Para poder resolver requerimientos de comportamiento asociados a los casilleros se modela una clase Equipo. Esta se utiliza para poder resolver los conflictos de inicialización de las unidades es sus respectivos casilleros. Ya que en el comienzo de la partida los jugadores sólo pueden colocar unidades es su mitad del tablero asignada. Esto se logra implementando el patron *double dispatch* para colocar unadades en casilleros del mismo equipo del jugador. A su vez existe en la fase de desarrollo de la partida una penalización de daño extra a unidades que son atacadas estando ubicadas en casilleros del bando contrario, por lo que también se implementaron métodos de la clase Equipo para resolver estas cuestiones aplicando este mismo patrón.

Con respecto a la clase AtaqueLejanoConExpansion utilizada como estrategia de ataque para la catapulta. Se sobreescribe el metodo atacar de la clase AtaqueLejano, pues además de atacar a una uidad en rango lejana debe impartir daño a todas sus unidades contiguas en el tablero. Para realizar este proceso se vale de las clases ConjuntoDeUnidades y ConjuntosDeUnidadesVictimas para encontrar a las unidades contiguas que debe dañar.

7. Diagramas de secuencia

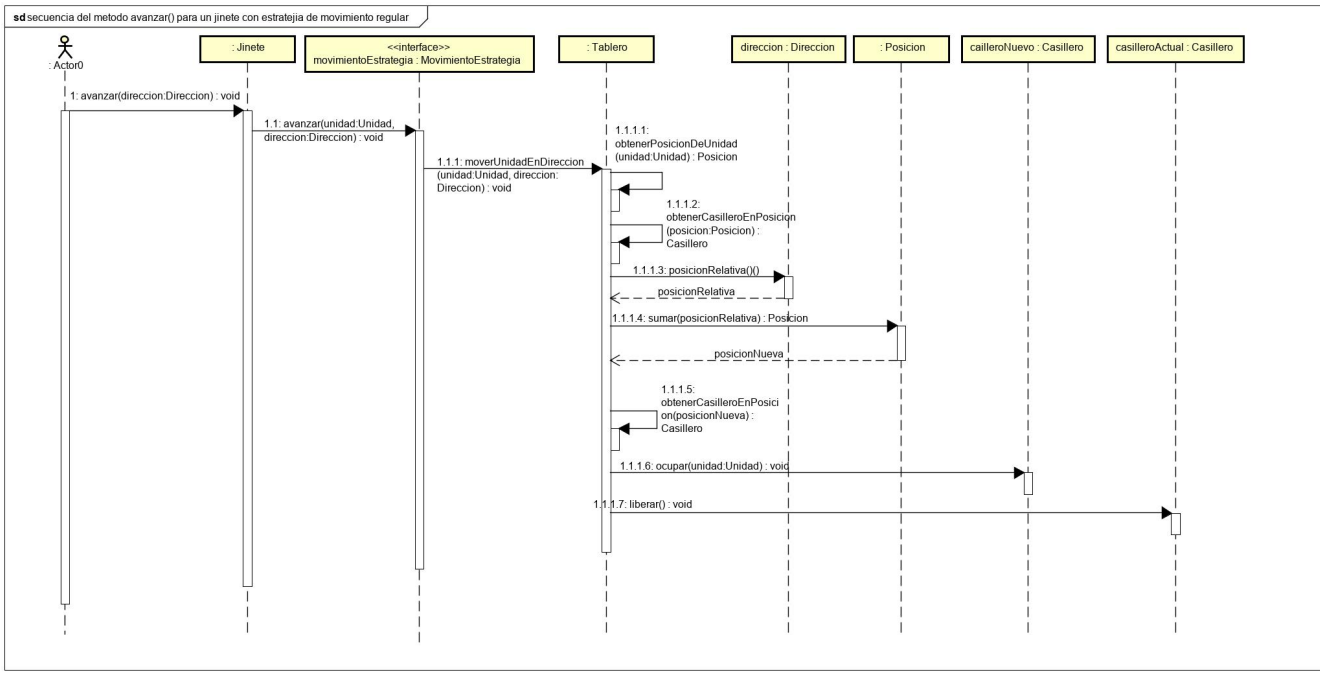

Figura 9. Diagrama de secuencia del método avanzar de un jinete con estrategia de movimiento regular.

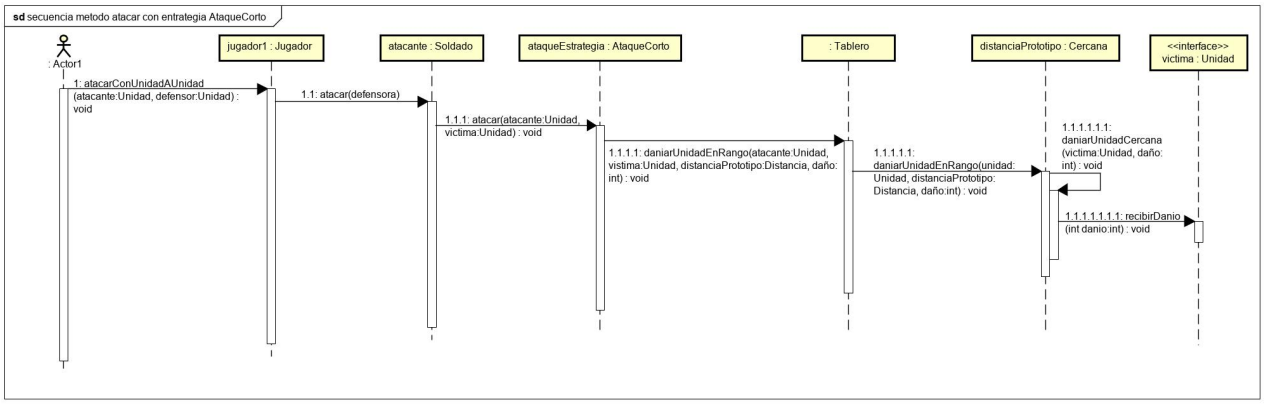

Figura 10. Diagrama de secuencia de método atacar con estrategia AtaqueCorto.

8. Excepciones

Algunas de las excepciones importantes son:

- *ProhibidoAtacarUnidadAliadaException:* caso en que se intenta atacar a una unidad que pertenece al mismo equipo.
- *CasilleroOcupadoException:* se utiliza en caso de que se quiera ocupar un casillero que ya está ocupado y por esto la unidad no se mueve.
- *CasilleroFueraDeTableroException:* para el caso en el que se quiera agregar una unidad fuera del rango permitido del tablero. Ésta excepción se atrapa en la clase Tablero cuando se ejecuta el método *obtenerCasilleroEnPosicion*().
- *CasilleroEsDeEnemigoException*: pertenece al a fase de inicializacion, caso donde se intenta colocar una unidad del bando contrario al del casillero seleccionado.
- *InsuficientePuntosRestantesAlColocarUnidadException*: caso en el que se selecciona una unidad de un valor mayor a la cantidad de puntos disponibles para elejir unidades del jugador.
- *ProhibidoCurarUnidadEnemigaException*: en el caso de la unidad curandero, esta solo puede curar unidades aliadas.
- *UnidadFueraDeRangoException*: caso en el cual se intenta atacar una unidad dentro de una distancia distinta a la estrategia de ataque de la unidad atacante.
- *UnidadNoPuedeMoverseException*: caso donde una unidad se intenta mover y no puede hacerlo, ya sea porque el casillero de destino este ocupado por otra unidad o porque la unidad seleccionada no puede moverse, como es el caso de la catapulta.