Trabajo Practico 2. Java

[75.06] Algoritmos y programación III

Curso 2

Segundo Cuatrimestre de 2019

Alumno	Padrón	Mail
Paz, Jonathan Nicolas	103340	jpaz@fi.uba.ar
Guastavino, Andres	103196	andresguasta@hotmail.com
Albornoz, Gonzalo	103554	gjalbornoz@gi.uba.ar
Codino, Federico	103533	fedecodino98@gmail.com

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo practico de la materia Algoritmos y Programación III que consiste en desarrollar una aplicación de un juego llamado Algochess. Este consiste en 2 jugadores los cuales cuentan con piezas con diferentes habilidades que se pueden atacar entre sí, y el jugador que logre eliminar todas las piezas del rival será el ganador.

2. Supuestos

El jugador es libre de decidir en que lugar del tablero posicionar las unidades. Si no cumple con los requisitos para un posicionamiento correcto (casilla ocupada o lado del tablero inválido) no perderá el turno, ni la unidad, ni los puntos que uso para comprar la unidad y se le avisará para que la pueda hacer un posicionamiento correcto de la unidad.

El jugador es libre de decidir a qué unidades atacar con la unidad que seleccione. Si no cumple con los requisitos (unidad objetivo aliada o unidad objetivo fuera de rango) no perderá el turno y se le avisará de su error para que pueda hacer un uso correcto de la unidad.

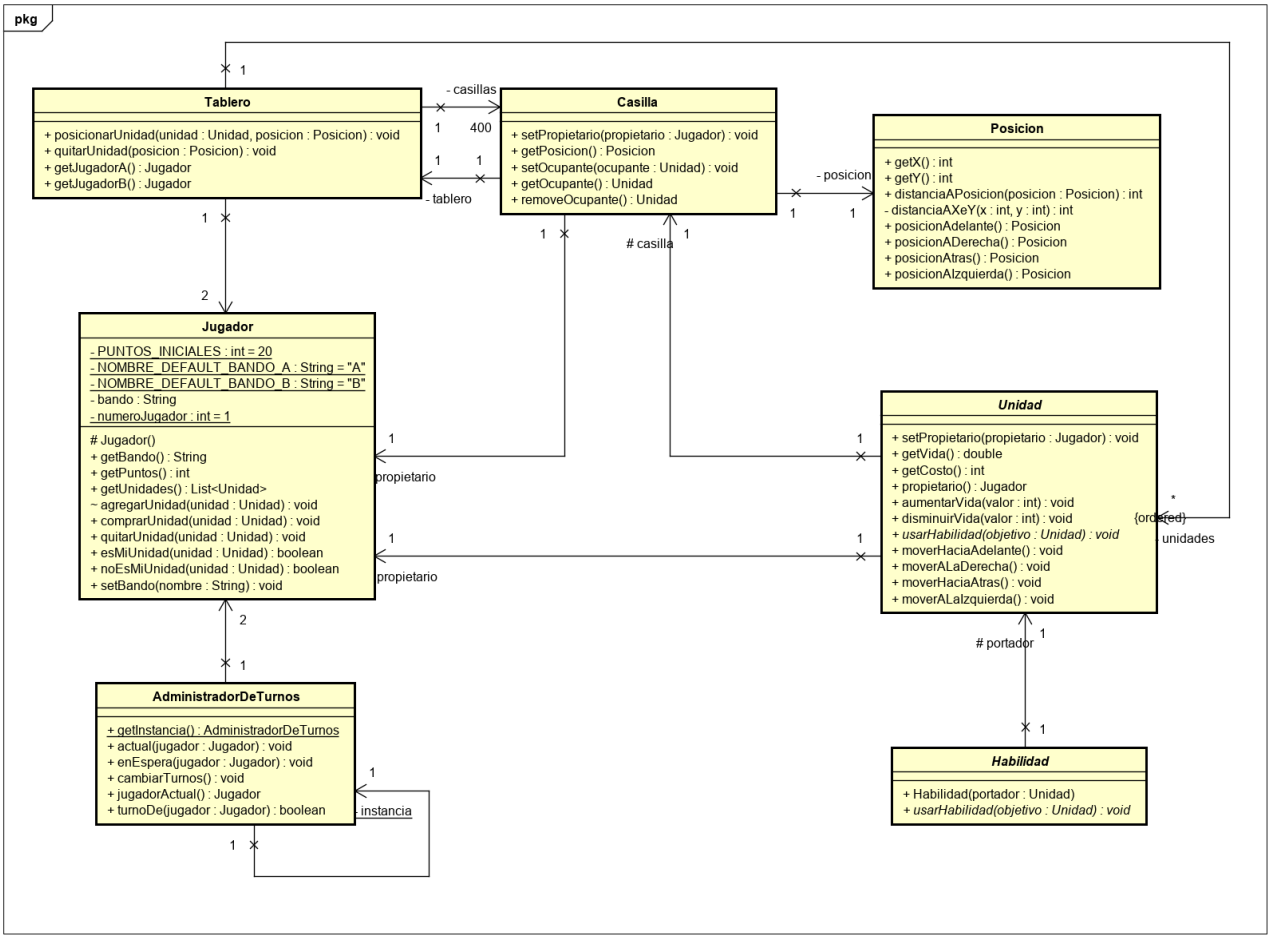
Las unidades del jugador 1 tendrán una imagen que ha de representarlos diferente de las del jugador 2, para discernir las unas de las otras.

Al jugador 1 se le asignará la mitad superior del tablero y al jugador 2 la mitad inferior.

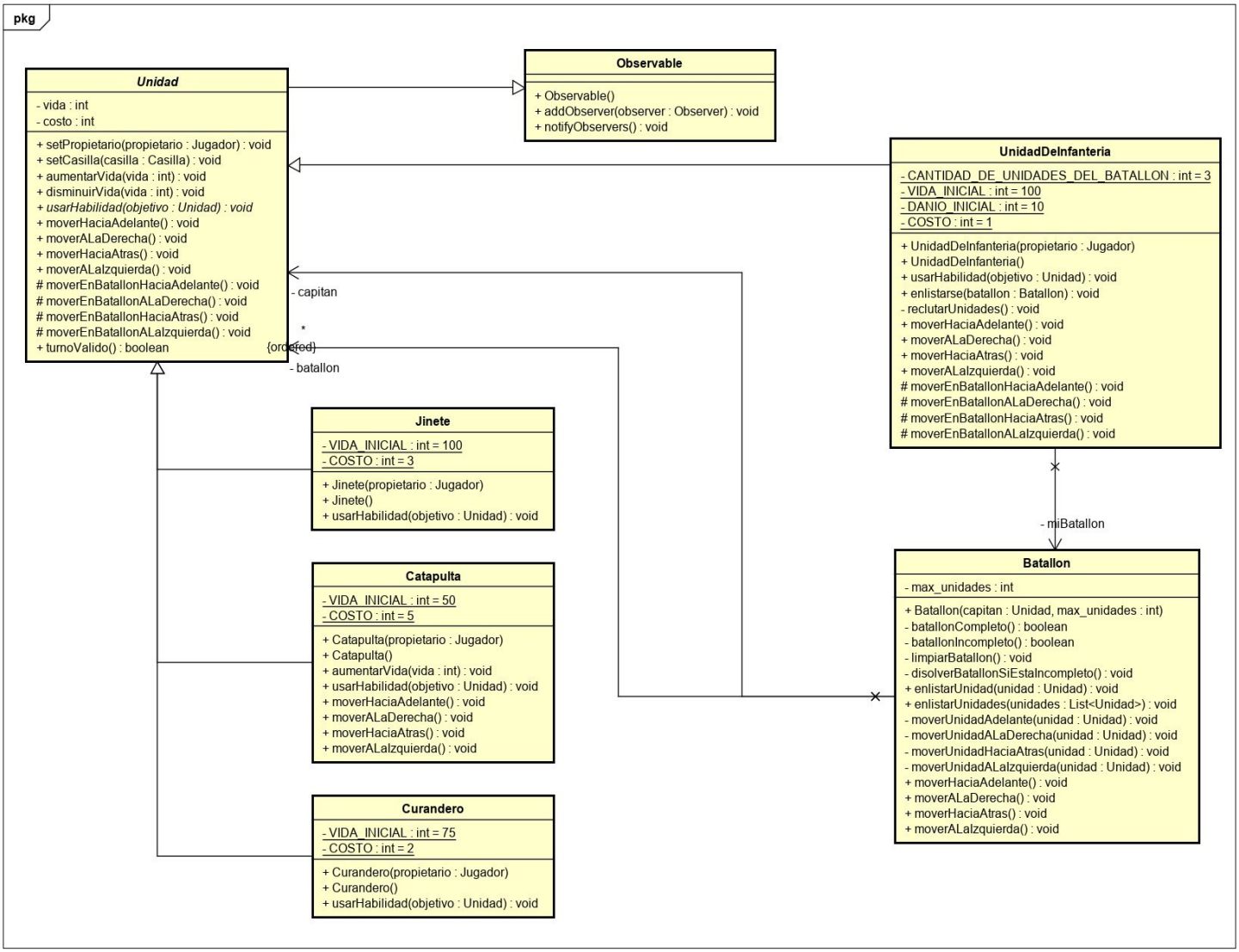
El jugador puede pasar el turno si desea no mover unidades.

Las unidades sólo pueden moverse adelante, atrás, a la derecha y a la izquierda de su posición actual.

Diagramas de clase y breve explicación de lo que realiza cada clase.



El diagrama de clases anterior es una representación general de las relaciones entre las diferentes clases que componen, a nuestra aplicación.



Dagrama de clase Unidad y sus relaciones

Unidad

Es el objeto el cual representa a una pieza en nuestro juego. Esta se puede mover y usar una habilidad. Además, cuenta con vida y tiene asociada una casilla, la cual representa su posición en el tablero, y un propietario , es decir a que jugador pertenece (en este caso puede ser jugadorA o jugadorB). Además, Unidad cuenta con una Habilidad y según esta variara el ataque de la Unidad.

Catapulta

La clase Catapulta es una Unidad por lo que también puede usar una Habilidad, en este caso cuenta con la Habilidad: AtaqueEncadenadoADistancia.
A diferencia del resto de las unidades esta no puede moverse por lo que si se le ordena a esta unidad que se mueva en cualquier dirección lanzara una excepción del tipo NoSePuedeMoverLaUnidad

Curandero

La clase Curandero representa a una Unidad la cual su habilidad es Curación y puede moverse en cualquier dirección.

Jinete

La clase Jinete hereda de Unidad por lo que esta se puede mover y utilizar una habilidad, en particular esta es AtaqueJinete

UnidadDeInfanteria

La clase UnidadDeInfanteria es una Unidad la cual puede moverse en cualquier dirección y cuanta con la Habilidad del tipo AtaqueACortaDistancia.
En especial, si UnidadDeInfateria se mueve y esta se encuentra rodeada de 2 UnidadDeInfanteria se mueven en forma de Batallon

Batallon

La clase Batallon no hereda de Unidad ya que es un conjunto de 3 UnidadDeInfanteria. Su función es mover de una en una las unidades pertenecientes al Batallon

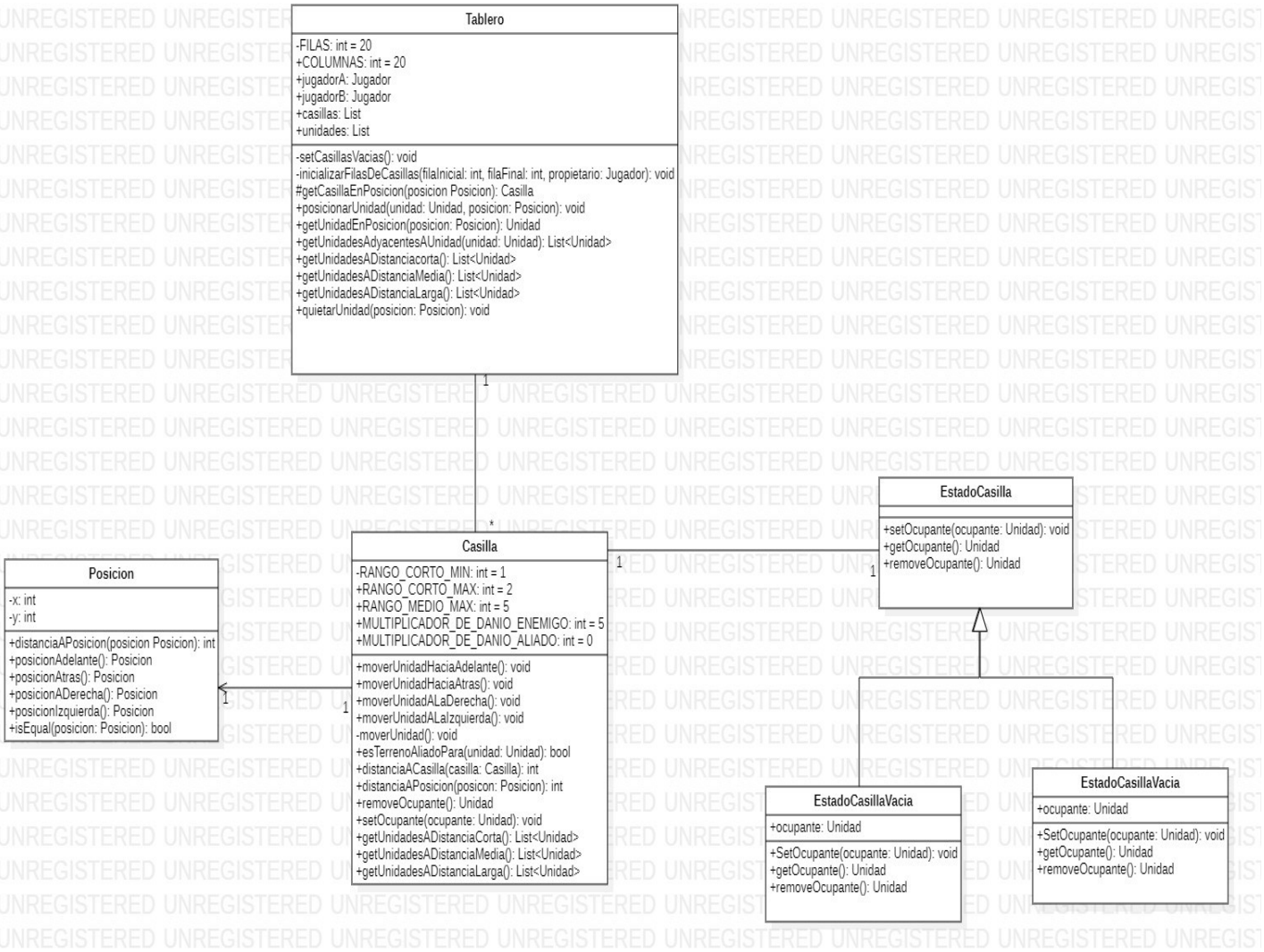


Diagrama de clase de Tablero y sus relaciones

Casilla

La clase Casilla es la encargada de mover la unidad a través del tablero. Además, esta cuenta con una referencia a la Unidad la cual se encuentra en la Casilla y a la Posicion de la Casilla.

También cuenta con una referencia al Jugador al que pertenece. Esto es de importancia ya que al comenzar el juego no se puede colocar una Unidad perteneciente al JugadorA en una Casilla perteneciente al JugadorB. Una vez finalizada la etapa de posicionamiento de unidades si una Unidad perteneciente al JugadorA avanza hasta una Casilla perteneciente al JugadorB la, esta recibe un 5% más de daño.

Por ultimo Casilla cuenta con una referencia a EstadoCasilla del cual dependerá si la Casilla puede ser ocupada o no.

EstadoCasilla

EstadoCasilla es una clase abstracta y de esta heredan EstadoCasillaOcupada y EstadoCasillaVacia.

EstadoCasillaOcuapada

La clase EstadoCasillaOcupada lanzara una excepción del tipo CasillaOcupadaException en caso de querer mover o colocar una Unidad en una Casilla ya ocupada.

EstadoCasillaVacia

La clase EstadoCasillaVacia aceptara a una Unidad para ocupará la Casilla y se encargara de cambiar su estado a EstadoCasillaOcupada

Tablero

La clase Tablero se encarga de inicializar las casillas que formaran al tablero asignándoles una Posición y un Propietario (una mitad del tablero pertenece al JugadorA y la otra al JugadorB) y a su vez se encarga de hacer contenedor para estas.

Posición

La clase Posición representa en que lugar se encuentra la Casilla a través de un vector 2D del estilo (X , Y) donde X e Y representan la fila y la columna en la que se encuentra la casilla.

Además, Posición es capaz de calcular sus posiciones adyacentes.

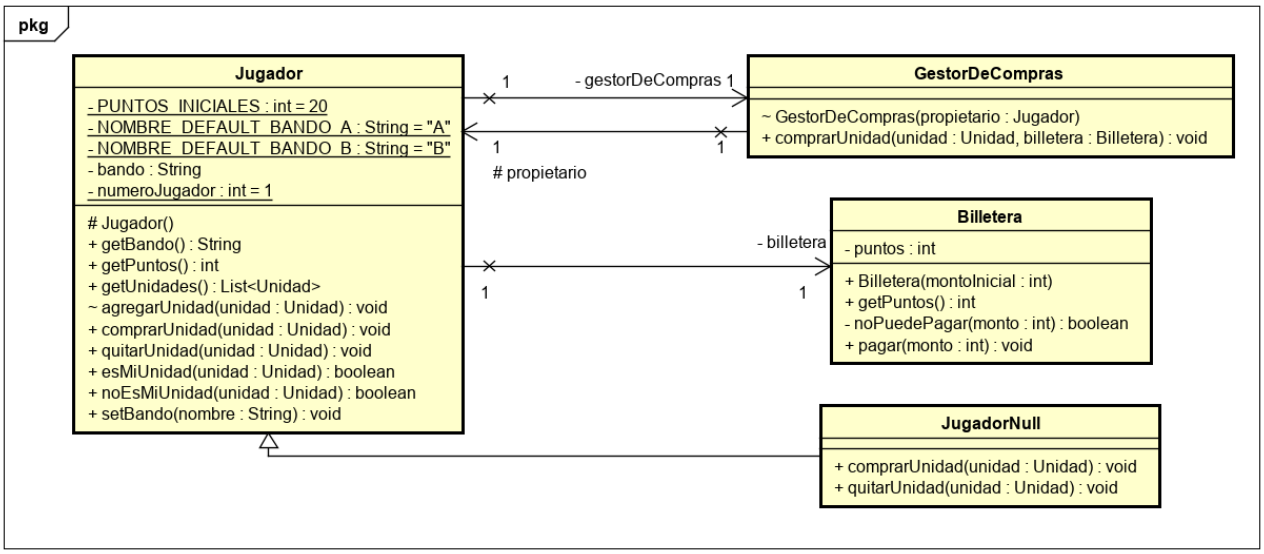


Diagrama de clase de Jugador y sus relaciones

Jugador

La clase Jugador es la encargada de realizar las Compras de las Unidades y dependiendo si un Jugador se quedo sin unidades perdió el juego. Además, jugador posee una referencia a GestorDeCompras, a Billetera y conoce las unidades que compró.

Cabe destacar que por default solo existirán 2 jugadores (“JugadorA” y “JugadorB”, ambos son strings representando los bandos), esto se debe a que según lo propuesto es un juego para 2 personas.

JugadorNull

La clase JugadorNull a diferencia de Jugador una vez indicado que compre una unidad esta entiende el mensaje pero no hace nada.

GestorDeCompras

GestorDeCompras es una clase abstracta la cual conoce a su propietario (un Jugador). Ante el mensaje comprarUnidad utilizará la Billetera (la cuál se pasa por parámetro) para realizar el pago y agregará la unidad a su propietario.

Billetera

Es la clase encargada de realizar los pagos por las Unidades que compra el Jugador.

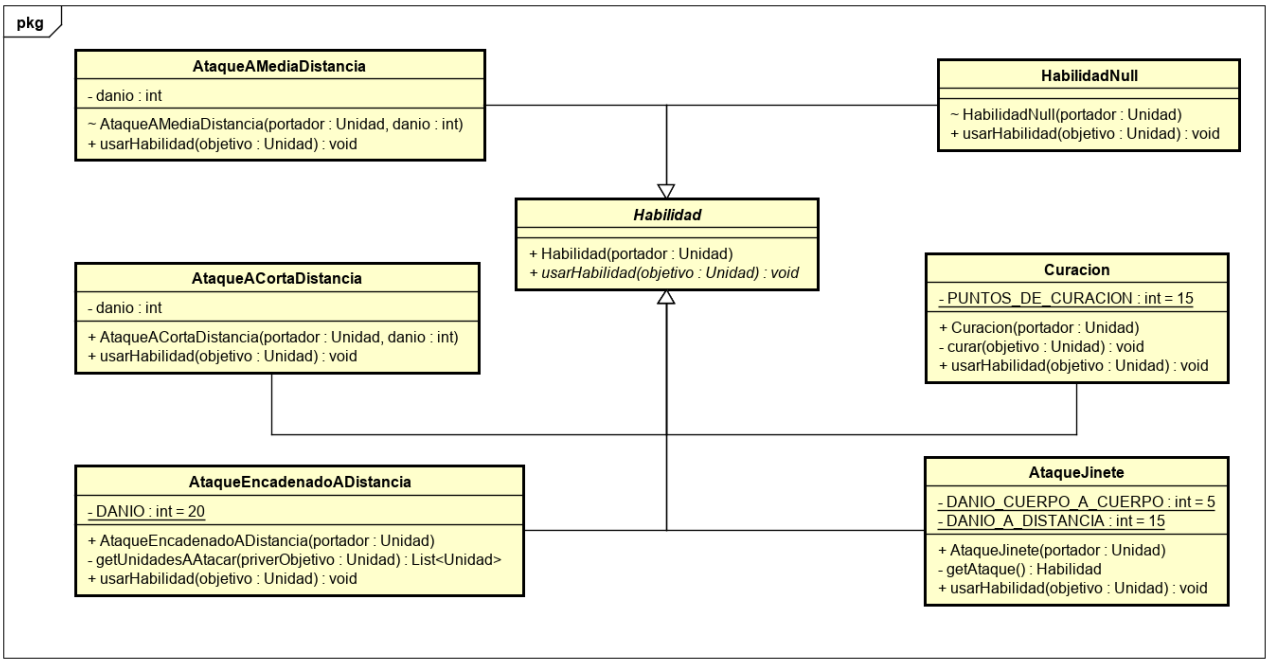


Diagrama de clase de habilidad y sus relaciones

Habilidad

Es una clase abstracta la cual tiene 6 clases hijas. Las Habilidades son usadas por las unidades para atacar.

Curación

Al pasarle una Unidad por parámetro se le cura 15 puntos de salud.

AtaqueJinete

Dependiendo de la posición y sus adyacentes de la Unidad que utiliza esta Habilidad cambiara el comportamiento.

Si hay al menos una UnidadDeInfanteria o no hay Unidades enemigas cercanas el ataque será un AtaqueAMediaDistancia.

Si no hay un ningun aliado cerca y hay enemigos cerca el ataque será del tipo AtaqueACortaDistancia.

Cualquier caso que no cumpla con ninguna de las dos condiciones anteriores el AtaqueJinete será del tipo HabilidadNull.

AtaqueEncadenadoADistancia

Al pasarle una Unidad por parámetro le hará daño a la Unidad y a sus Unidades adyacente.

AtaqueACortaDistancia

Si la Unidad seleccionada para ser atacada se encuentra a una distancia corta del atacante se le restara vida a la Unidad seleccionada, caso contrario se lanzara una Excepción del tipo UnidadFueraDeRango.

AtaqueAMediaDistancia

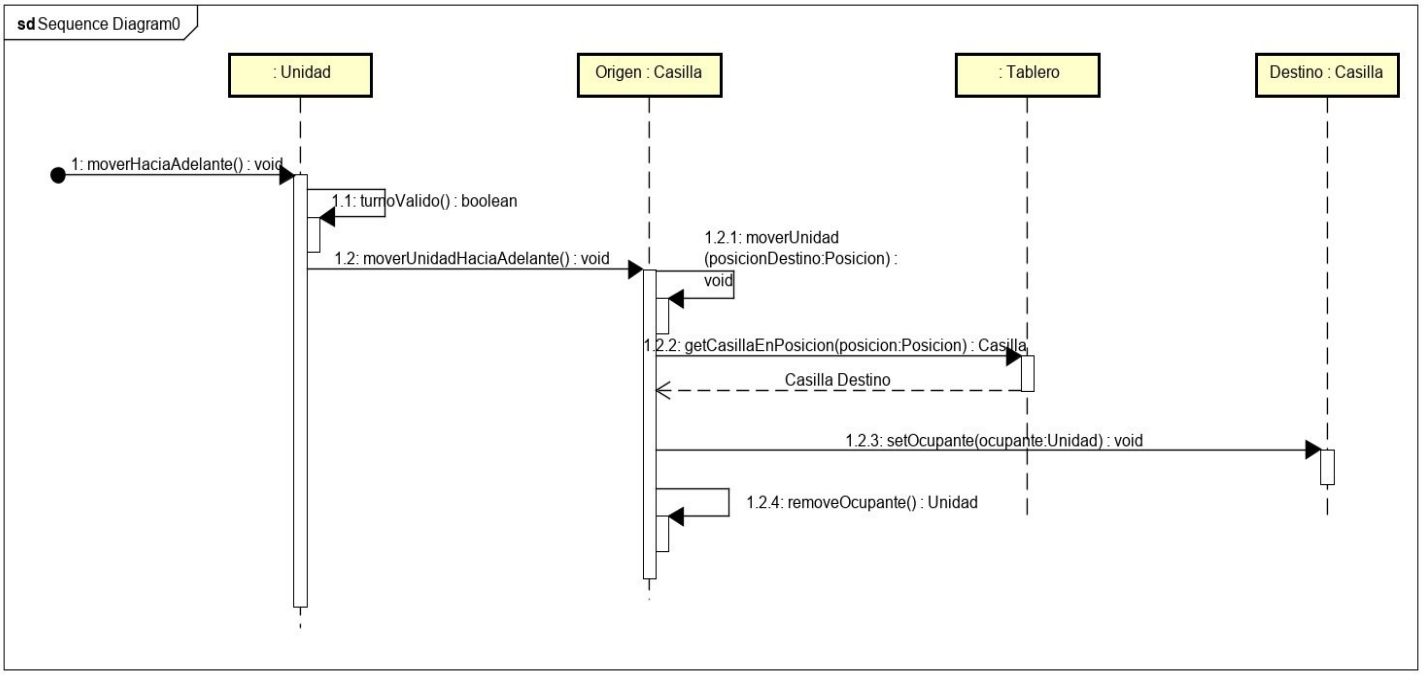
Si la Unidad seleccionada para ser atacada se encuentra a una distancia media del atacante se le restara vida a la Unidad seleccionada, caso contrario se lanzara una Excepción del tipo UnidadFueraDeRango.

HabilidadNull

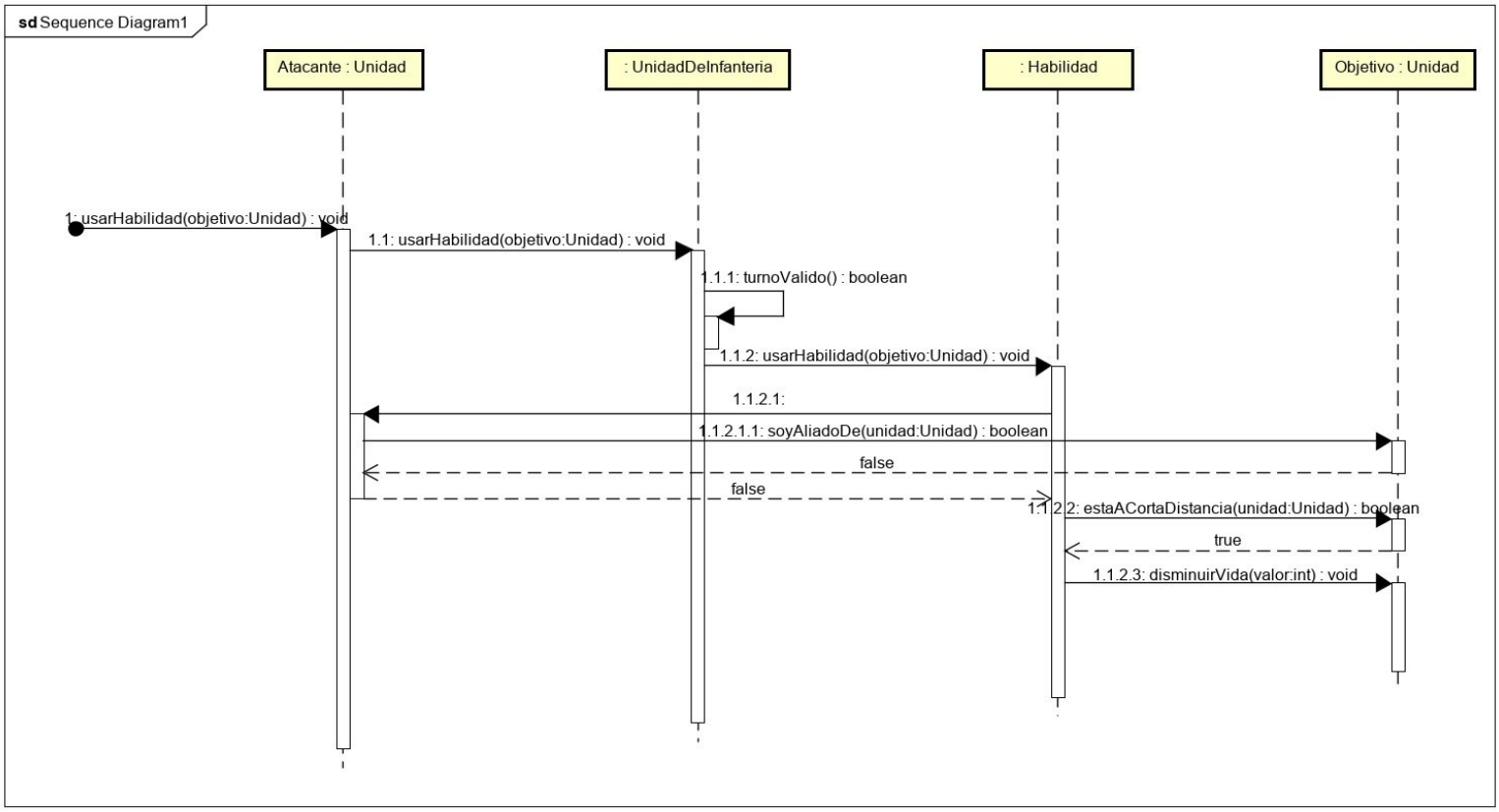
Es una clase la cual no realiza ninguna acción, es decir, entiende los mensajes pero no realiza nada.

Diagramas de secuencia

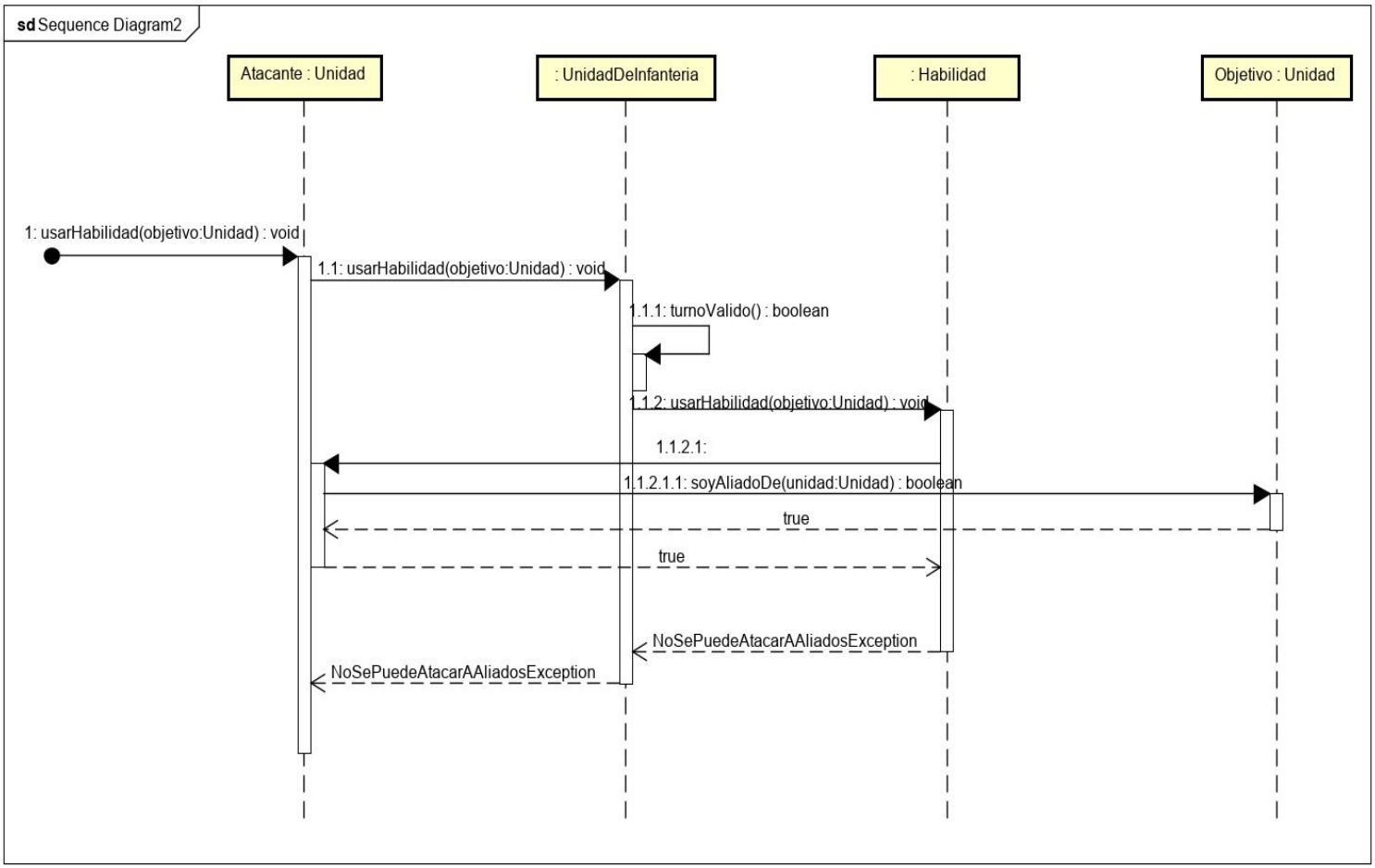
A continuación presentaremos los diagramas de secuencia que presentan una mayor importancia en nuestro modelo.



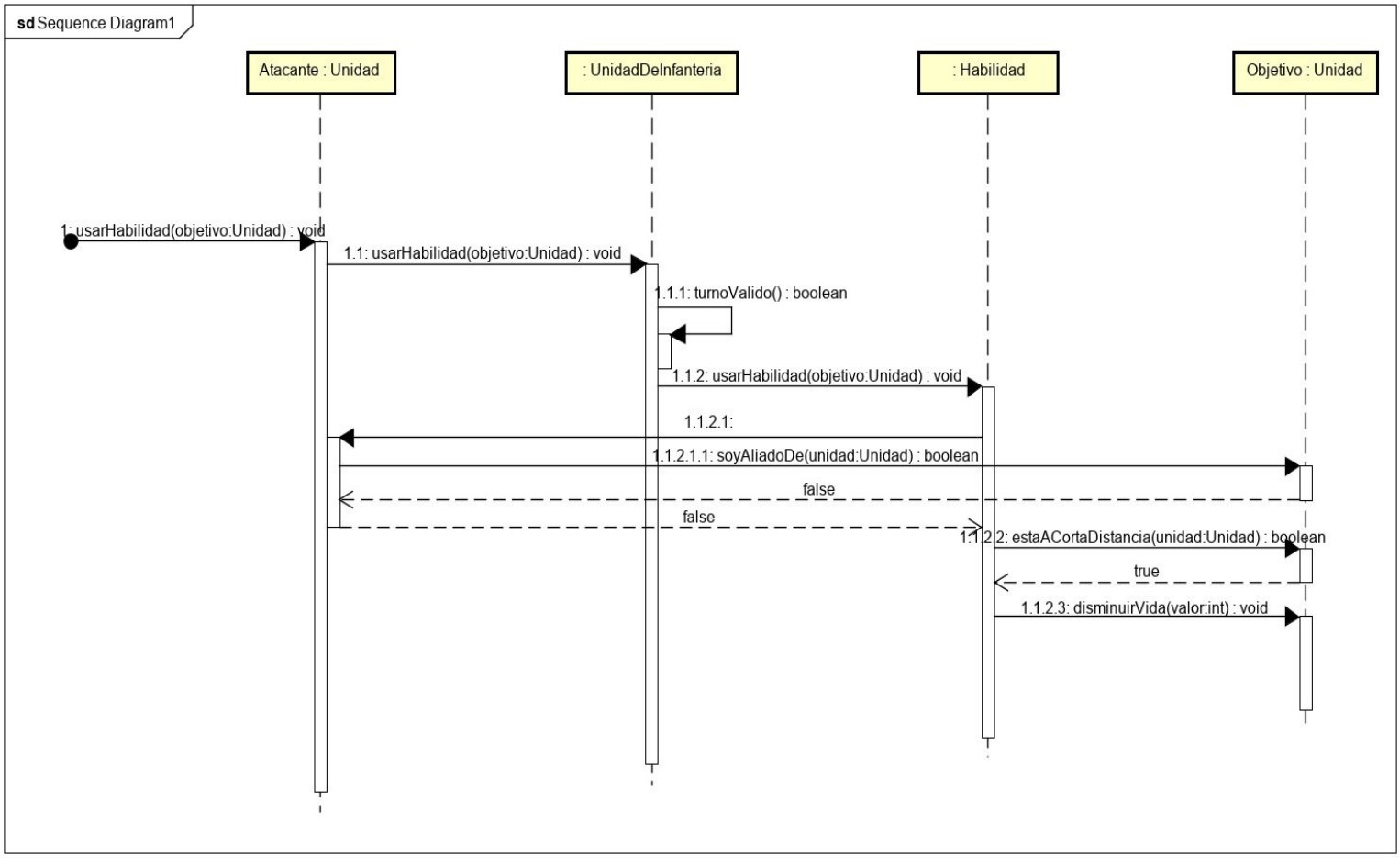
En el diagrama anterior podemos observar cómo es el intercambio de mensajes a la hora de que se mueva una Unidades



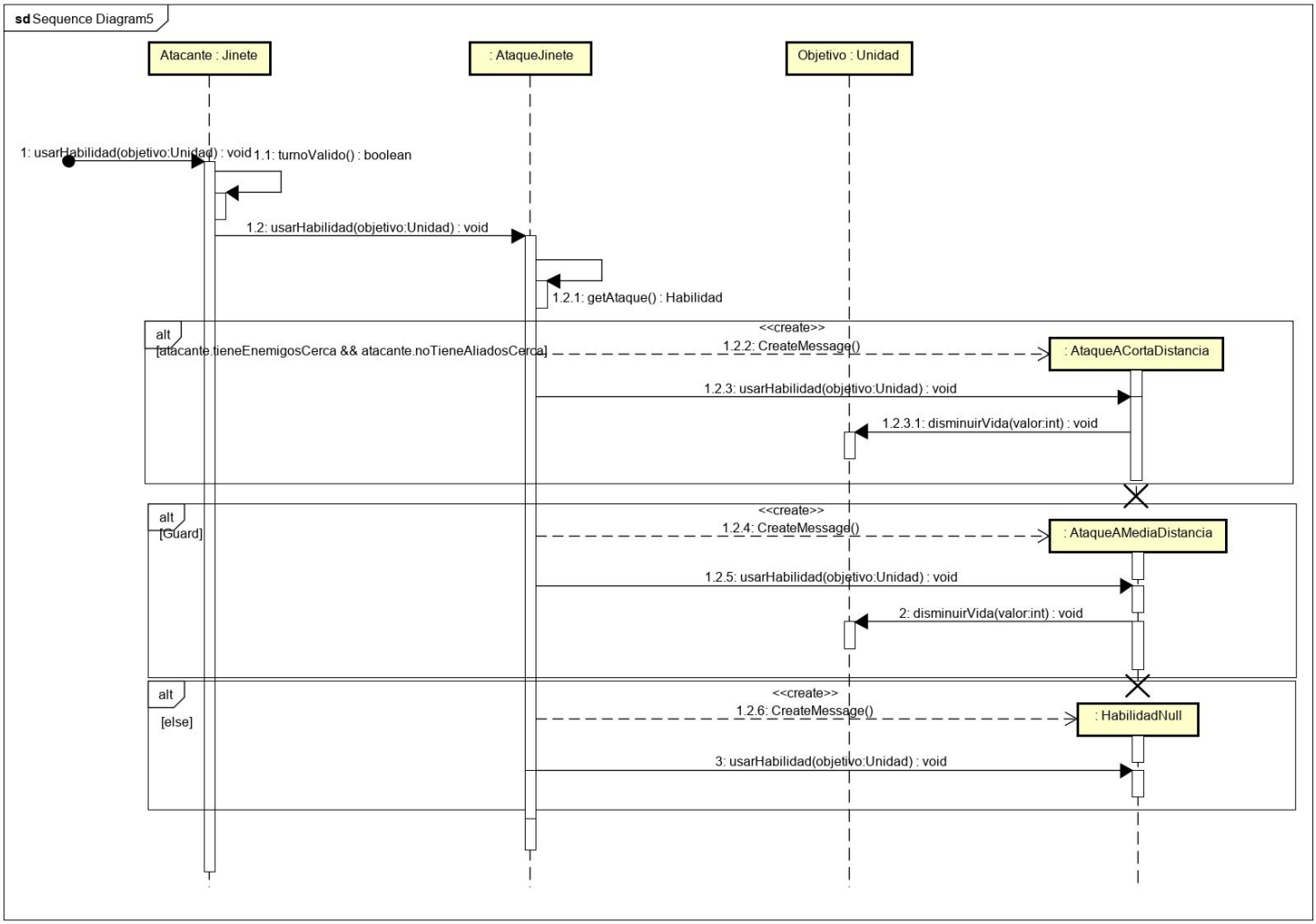
En el diagrama percibimos como actuario el código a la hora de querer ataca una unidad enemiga con una UnidadDeInfanteria. Si bien en el diagrama se utiliza una UnidadDeInfanteria el proceso ocurre de igual manera para cualquier otra unidad, sólo cambia el tipo de ataque que realiza.



El diagrama muestra cómo funciona el método usarHabilidad en el caso en que se quiera usar la habilidad con una unidad aliada.

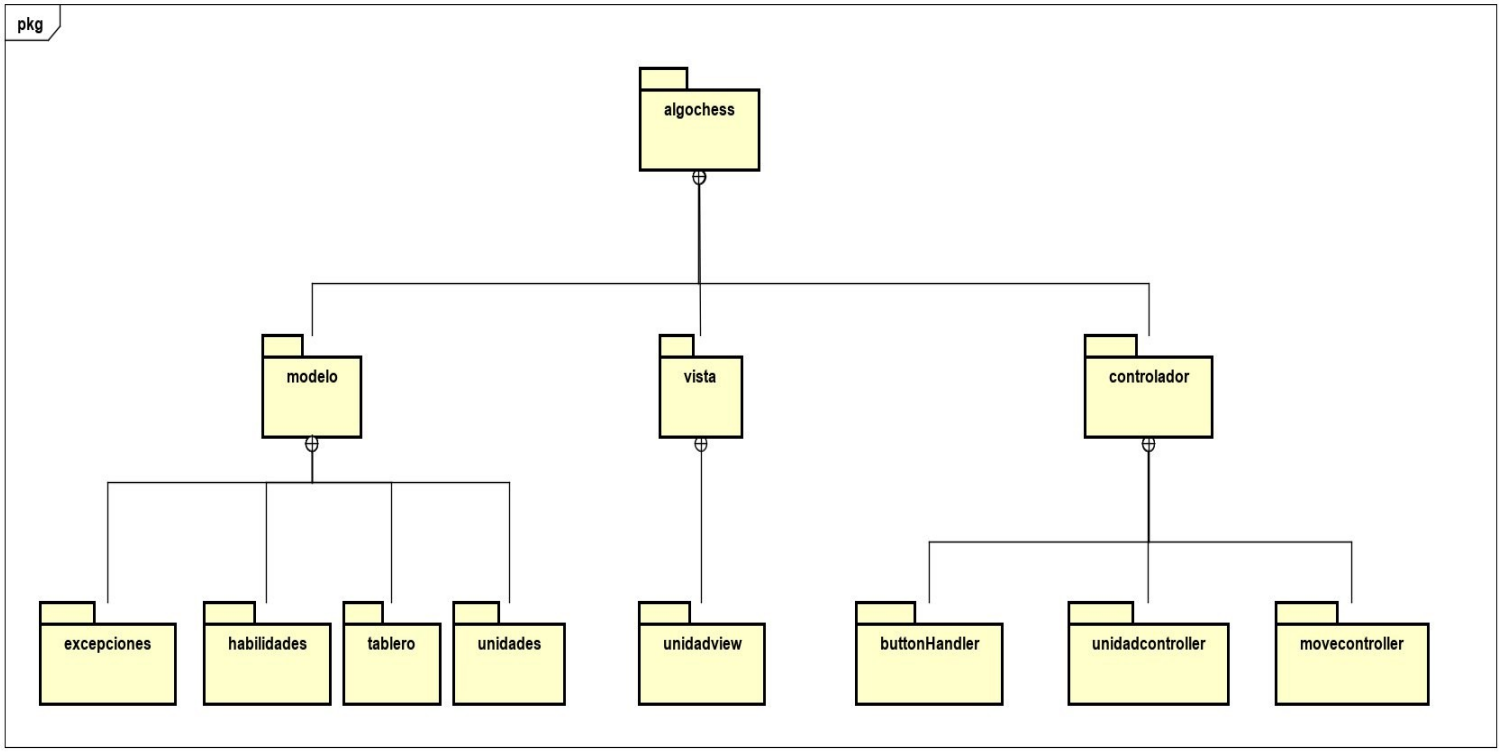


El diagrama nos enseña como responderá el código ante la solicitud de querer usar el método usarHabilidad y la unidad enemiga se encuentra fuera de rango.



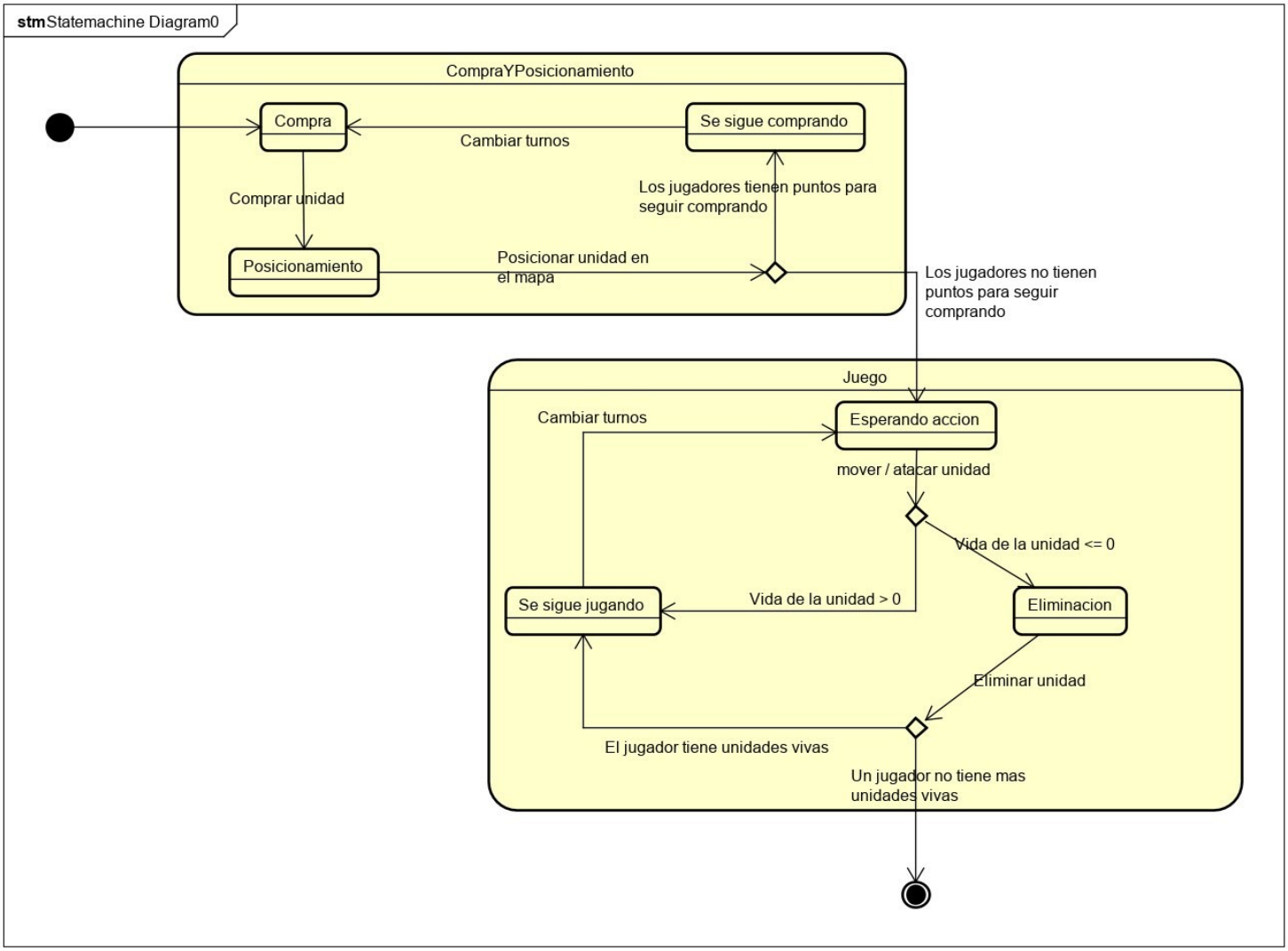
El diagrama muestra cómo actúa el código a la hora de querer usar el método usarHabilidad del jinete, presentando sus diferentes casos.

Diagrama de paquetes



El diagrama de paquetes nos enseña cómo está dividido nuestro sistema en diferentes carpetas. Es importante aclara que solo decidimos incluir las diferentes carpetas sin sus clases ya que esto empeoraría la legibilidad del diagrama y los diagramas de clases ya están incluidos en el informe.

Diagrama de Estado



El diagrama nos presenta cual es el estado por el que pasa el juego desde que se inicia hasta que este termina.

Detalles de Implementación

Como podemos observamos en el diagrama de Clase de Unidad, Catapulta, jinete, UnidadDeInfanteria y Curandero heredan de Unidad, esto se debe a que las 4 clases mencionadas al principio tienen tanto métodos como atributos en común. Por lo que nos ahorramos volver a escribir los mismos métodos en diferentes clases mejorando la legibilidad del código y en caso futuro de tener que modificar algún método que tienen en común (por ejemplo: “disminuirVida”) simplemente modificaremos a la clase madre. Además, cabe destacar que Catapulta, Jinete, UnidadDeInfanteria y Curandero cumplen la relación “es un”, es decir, Catapulta, Jinete, UnidadDeInfanteria y Curandero “es un/a” Unidad.

Como el diagrama de clase de Habilidad lo indica, allí también decidimos utilizar herencia esto se debe en parte a los argumentos mencionados en el párrafo anterior y en otra parte a que una Unidad puede tener una habilidad y esta puede ser Curación, AtaqueJinete o cualquiera de las clases hijas de Habilidad, por lo tanto decidimos que aplicar herencia en este caso sería lo mas conveniente para facilitar la implementación del código.

Es importante mencionar que para resolver los problemas presentados por el enunciado implementamos el uso de patrones de diseño. En especial el uso del patrón **NullPattern** el cual utilizamos para implementar HabilidadNull y JugadorNull, esto se debe, a que queríamos que tanto HabilidadNull como JugadorNull entendieran los mensajes de sus clases madres pero que estas no respondan, es decir, no hagan nada. También en el controlador de movimientos de unidades se implementa el MovimientoControllerNull.

Otro de los patrones implementados es el patrón **Factory**, implementado para la creación de vistas en ViewFactoryA, ViewFactoryB, y para la creación de controladores para las vistas UnidadControlFactory. Ambas Factory’s se basan en un Map que mapea, en el caso de las vistas, la clase de una unidad que recibe por parámetro con su respectiva vista. Así, si la Factory recibe una UnidadDeInfanteria creará y retornará una UnidadInfanteriaView. Si en ese momento es el turno del jugadorA se utilizará la ViewFactoryA y se devolverá una UnidadInfateriaAView. Si es el turno del jugadorB se utilizará la ViewFactoryB y se devolverá una UnidadInfateriaBView, y similar para las demás unidades. Para los controladores de la vista sucede lo mismo, UnidadControlFactory recibe una unidad y dependiendo de la clase de esa unidad retornará el correspondiente controlador para la vista.

Otro patrón del que hacemos uso es el patrón **State**. Un ejemplo de la aplicación de este patrón es en la clase Habilidad, más específicamente en el AtaqueJinete, el cuál cambia entre las habilidades AtaqueCortoAlcance, AtaqueMedioAlcance y HabilidadNull dependiendo de las condiciones en las cuáles se le solicite a la unidad que lo porta usar su habilidad. Otra implementación de este patrón se puede ver en el EstadoCasilla, el cuál cambia entre EstadoCasillaOcupada y EstadoCasillaVacia a lo largo de la ejecución del programa según tenga una unidad o no, y dependiendo si la unidad se mueve o no.

También se implementa el patrón **Proxy**. Su implementación se puede ver en la clase ChatBoxProxy, el cual guarda una referencia a ChatBox, mediando entre el usuario y el ChatBox.

Patrón **Singleton**. Hacemos uso de este patrón de diseño en la clase AdministradorDeTurnos, debido a que es utilizada en distintas partes del programa pero como lleva la cuenta de los turnos es necesario que haya una sola instancia de la misma y que sea de acceso global. Otra implementación

en el código de este patrón es en AccionesController. En este caso esta clase fue creada para manejar los movimientos de las unidades, así como los ataques. Debido a su comportamiento era necesario que también fuera de acceso global.

Excepciones

Las excepciones que se explicaran a continuación todas heredan de RunTimeExcepction

BatallonCompleto : se utiliza a la hora de formar un Batallón y este está completo, se la atrapa en el mismo batallón.

CasillaOcupadaException: su propósito es ser lanzada cuando se quiere posicionar o mover una Unidad a una Casilla que se encuentra ocupada.

FinDelJuegoException: esta excepción será lanzada cuando un jugador se quede sin piezas indicando que se terminó el juego.

NoExisteLaCasillaEnPosicionException: esta excepción se lanza al enviarle el método getCasillaEnPosicion al tablero con una posición invalida.

NoSePuedeAtacarAAliados: se lanzará esta excepción cuando una unidad intente atacar a otra pero ambas pertenecen al mismo jugador.

NoSePuedeMoverLaUnidad: se lanzará esta excepción cuando se intente mover una unidad de tipo Catapulta.

NoSePuedePosicionarEnTerrenoEnemigo: se lanzará esta excepción cuando se intente posicionar una unidad del jugador1 en la mitad del tablero del jugador2 o la inversa.

NoSePuedeUnirABatallonRival: se lanzará esta excepción cuando se quiera reclutar a una unidad de infantería enemiga para armar un batallón de unidades de infantería.

PuntosInsuficientesException: se lanzará esta excepción cuando se intente comprar una unidad pero el jugador no posea puntos suficientes para comprarla.

UnidadFueraDeRango: se lanzará esta excepción cuando se intente atacar a una unidad que no esté en rango del ataque de la unidad atacante.