FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — AlgoChess

[7507/9502] Algoritmos y Programación III
Curso 2
Segundo cuatrimestre de 2019

Nombre y apellido	Padrón
Juan Ignacio Colombo	103471
Francisco Gutierrez	103543
Tobias Kleppe	102545
Nicolás Salvanescki	100638

Índice

1. Introducción

El informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III, que consiste en desarollar un juego en java. El juego consiste en dos jugadores, que colocan en un tablero unidades, con el fin de destruir las del enemigo contrario. Los jugadores tienen un determinado numero de puntos que pueden gastar para colocar a las unidades. El juego termina cuando un jugador se queda sin fichas, siendo este el perdedor.

2. Supuestos

En el juego las piezas arrancan con determinada vida. Una pieza disponible es el Curandero, el cual puede curar a las piezas. Al curar a algunas piezas estas no tienen limite de vida maxima, ya que consideramos como parte de la estrategia del juego, la habilidad de saber utilizar bien las habilidades especiales de cada pieza.
Al crear una unidad, el jugador debe escoger la posición en el tablero donde quiera que esté la misma. Es decir, el proceso de crear unidades y el de ubicarlas en el tablero se da al mismo tiempo. Un jugador no puede crear una unidad sin haber especificado donde quiere que ésta sea creada.

3. Diagramas de clase

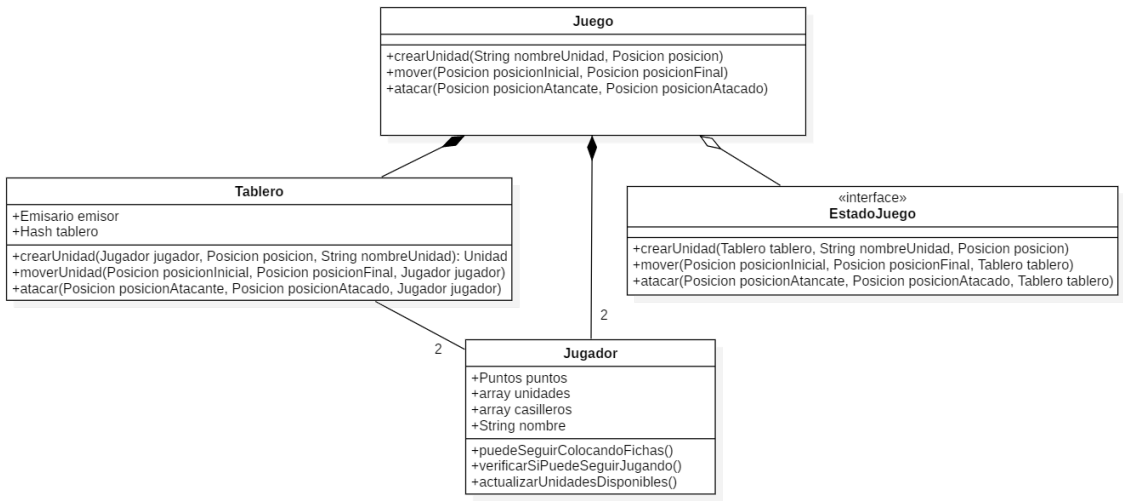

Figura 1: Diagrama de clases de Juego.

En la Figura 1, podemos ver el diagrama de clases del Juego. En el mismo se maneja en primera instancia el flujo del sistema de turnos, además de servir como mediador entre el controlador y el modelo. La clase cuenta con una referencia al tablero, otra a ambos jugadores que se encuentran en la partida y una instancia de EstadoJuego, que es una interfaz de la cual se hablará mas en profundidad en la Figura 2. Juego cuenta con trés métodos principales, CrearUnidad, mover y atacar.

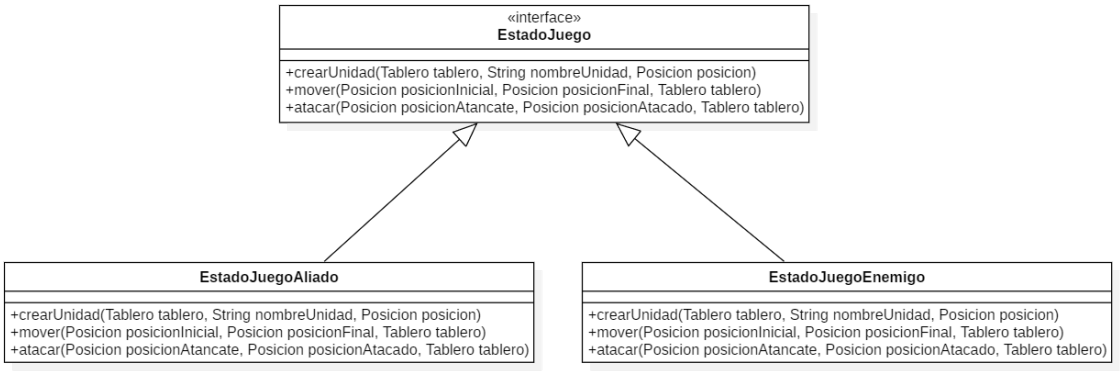

Figura 2: Diagrama de clases del EstadoJuego

En la Figura 2, podemos ver el diagrama de clases de EstadoJuego. El mismo conciste en una interface cuyos hijos son EstadoJuegoAliado y EstadoJuegoEnemigo, que justamente son los que modelan los turnos, cuando corresponde al "jugadorAliadoz cuando al "jugadorEnemigo- espectivamente. Ambas clases hijas cuentan con una referencia al jugador y al JugadorRival

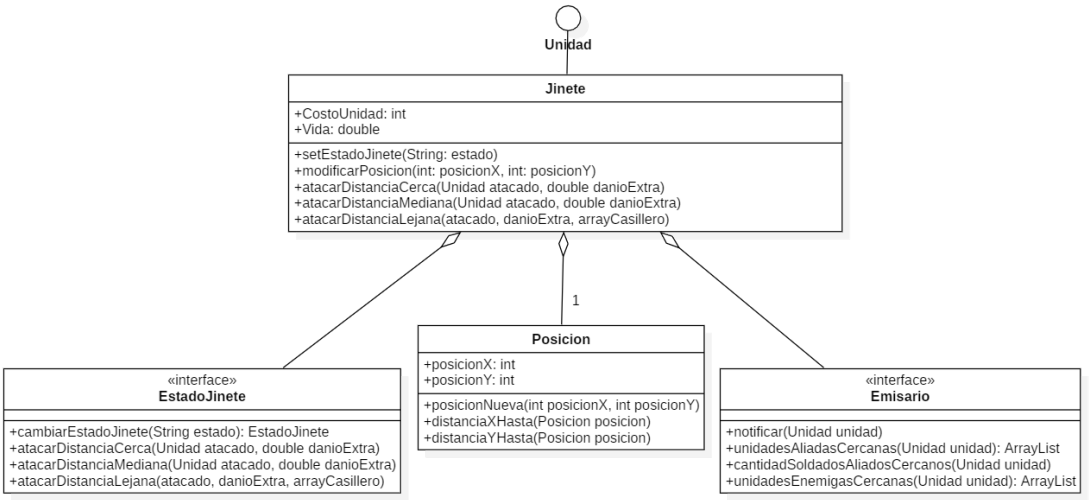

Figura 3: Diagrama de Clases de Jinete.

En la figura 3 se observa el diagrama de clases del jinete. El mismo está conformado por su Estado, del cual se habla en profundidad en la Figura 3, una Posicion y la un Emisario, cuya función es brindar un intermediario entre el tablero y la unidad, en la Figura 4 se vé el diagrama de Clases del mismo. El jinete también cuenta con vida y su costo, al igual que con los métodos heredados de la clase madre, Unidad (polimorfismo).

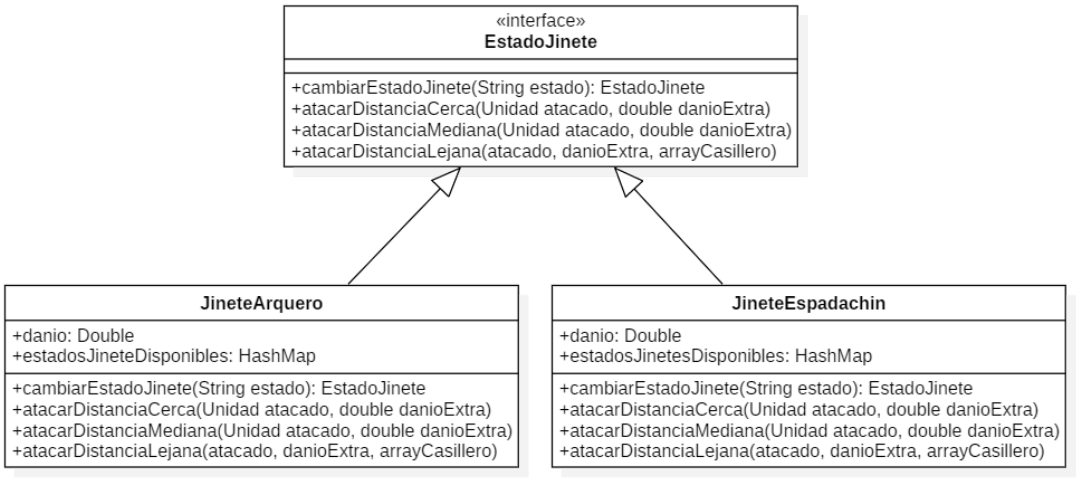

Figura 4: Diagrama de Clases de la interfaz EstadoJinete.

En la figura 4 se denota la implementación de la interface EstadoJinete, el objetivo de esta es poder manejar mediante el patron State las variaciones del estado de jinete que pasa a ser un JineteEspadachin, (hay enemigos cercanos), o un JineteArquero (no hay enemigos cercanos)

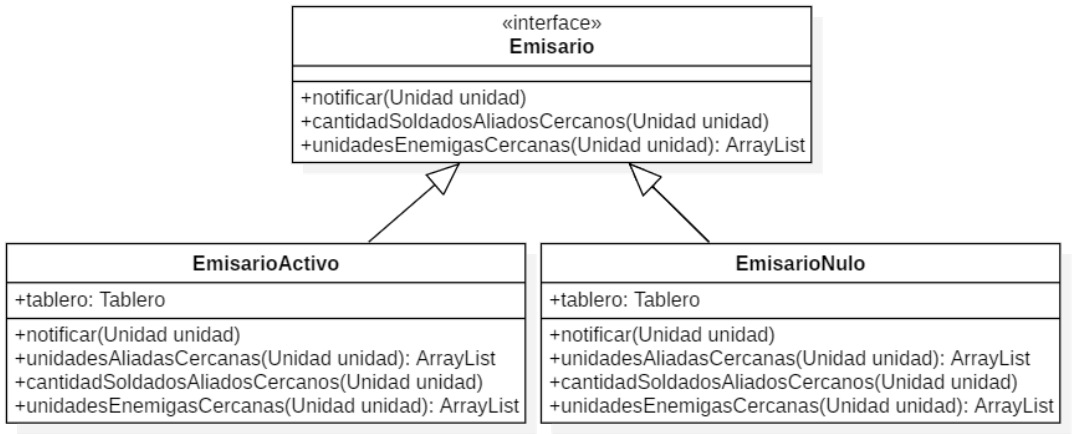

Figura 5: Diagrama de Clases del Emisario.

La figura 5 muestra el diagrama de clases del Emisario, el mismo cuenta con un Null object pattern, ya que mediante polimorfismo una instancia de Emisario puede ser EmisarioReal o un EmisarioNulo, que el mismo sería el nullObject en cuestion, es decir, el no Emisario. Esta clase Emisario sirve para notificar al tablero de interacciones dadas por la unidad al tablero, o viceversa, en el caso del jinete como necesita saber que unidades cercanas hay, una interacción es que en tiempo de ejecución este objeto le comunique si hay o no enemigosCercanos al Jinete.

4. Detalles de implementación

En el presente trabajo, se destacó el uso de los patrones de diseño vistos en clase, así como también la utilizacion de los principios SOLID, a continuación se detalla como se dió la implementación de ambos en el modelado de clases:

Null Pattern: se agrega la no unidad ÜnidadNula"mediante el patrón de diseño "null paternçon el objetivo de denotar la instancia en la que se podría encuentrar un casillero, por ejemplo, al estar vacio, el mismo no tiene ninguna unidad colocada sobre él, por ello, es necesario contar con esto en nuestro modelo mediante la implementación de esta clase ÜnidadNula", es decir la no unidad. También a la hora de utilizar emisarios, notamos que en algunos casos no necesitabamos la precencia del mismo, y por ello fue fundamental crear al .ºEmisarioNulo", que simplemente recibe mensajes pero no reacciona a los mismos, respetando el comportamiento de este patrón.

State: Para "saber"si un casillero esta ocupado o vacío se implementó con este patrón el, EstadoCasillero que puede ser EstadoCasilleroVacio o EstadoCasilleroOcupado. También como se vé en la Figura 2, el mismo patrón se utilizó en EstadoJuego, a la hora de modelar las unidades, fué útil en el momento de crear un "JineteEspadachin.º un "JineteArquero", ver figura 4

Factory: A la hora de crear unidades necesitamos de una fábrica que cree los mismos, y este patrón es perfecto para ello.

MVC: Fué utilizado para separar el trabajo en tres partes clave, el Modelo, la Vista y el Controlador, la vista conoce al modelo y al controlador, el controlador al modelo y a la vista, mientras que el modelo no conoce a ninguno. Esto nos sirve para que haya un flujo mas ordenado en la comunicación de las partes, para que si el día de mañana se quiera dar otro enfoque a la vista del programa, por ejemplo, quiera ser ejecutado en una pagina web, esto se pueda hacer sin mucho problema. Lo dicho respeta ampliamente el principio open closed

Reflection: Fué utilizado repetidas veces a la hora de modelar la vista por que si algún principio debe ser ïnfringido"decidimos que lo óptimo sea desde la vista, ya que esto respeta mas el principio open closed, como en el ejemplo que dimos en el ítem anterior.

5. Excepciones

CurarException: El proposito de esta excepcion, es que cuando en el juego el curandero trate de curar a una Catapulta, se lanze la excepcion, ya que uno de los impedimentos de la catapulta es la de no poder ser curada.

NoAlcanzanLosPuntosException: Es lanzada cuando el jugador está tratando de crear una unidad sin tener puntos suficientes para hacerlo. De esta forma se evita crear más unidades de las que son posibles

NoPuedeAtacarException: El proposito de esta excepcion, es que cuando una unidad que no pueda atacar, como por ejemplo el soldado no puede atacar a distancia, o el curandero que no puede atacar, se lanze esta excepcion.

UnidadInvalidaException: El proposito de esta exceptio, es que cuando el jugador quiera crear una unidad que no exista, se lanze esta excepcion.

UnidadNulaException: Cuando se quiere utilizar una pieza de UnidadNula como si fuera una unidad cualquiera del juego lanza la siguiente excepción.

MovimientoInvalidoException: Cuando se quiere mover una unidad de forma ilícita lanza la esta excepción.

CasilleroEnemigoException: Cuando se quiera crear una unidad esta excepción se lanza si quiere ser creada en un casillero enemigo.

CasilleroOcupadoException: Si se quiere mover una unidad a un casillero ocupado se lanza esta excepción.

CasilleroVacioException: Si se quiere obtener o eliminar una unidad de un casillero vacío.

FaseCreacionUnidadesFinalizoException: Si se quiere crear una unidad una vez finalizada la fase de creación de unidades se lanza esta ex.

JugadorPerdioException: Se lanza si el jugador perdió

JugadorSeQuedoSinPuntosException: Se lanza si el jugador no tiene mas puntos

6. Diagramas de secuencia

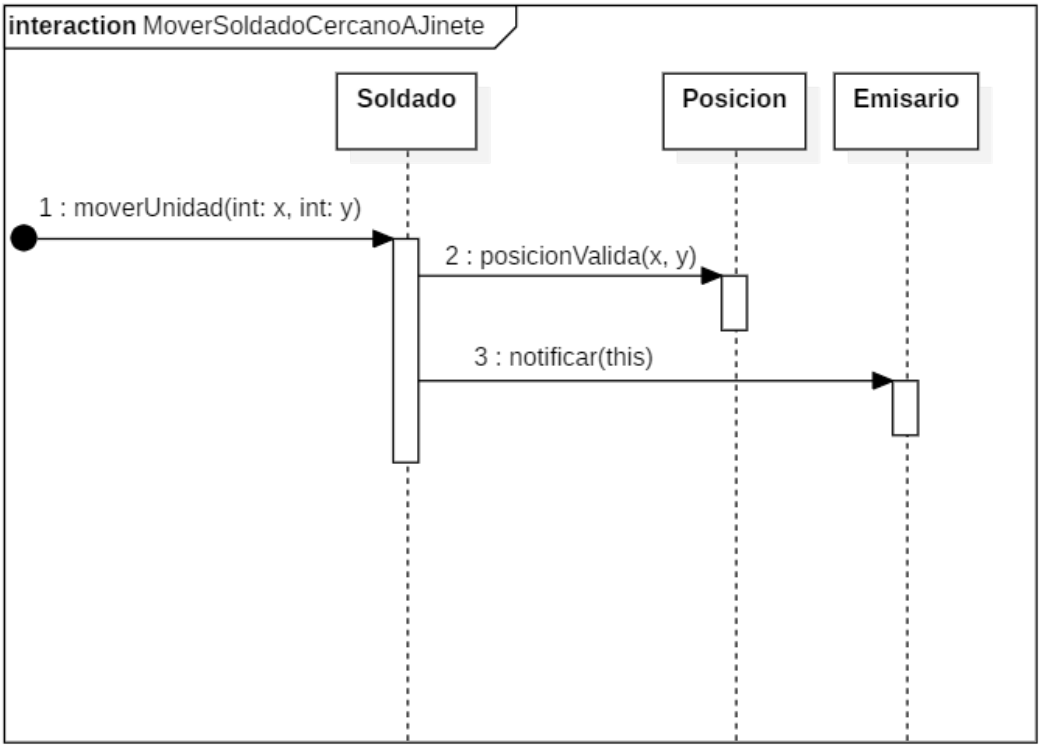

Figura 6: Diagrama de Secuencia al mover un soldado.

El diagrama de la Figura 6 muestra el diagrama de secuencia de como se producen las relaciónes entre clases dado por el flujo del modelo a la hora de moverUnidad, en este caso preciso se está moviendo un Soldado a una posición cercana a un Jinete. Elegimos mostrar este caso dado que tiene varios conflictos a la hora de resolverse, porque no solo hay que mover el soldado sino que debe cambiar el estado del jinete (proximos diagramas). Como se vé, el soldado verifica mediante Posicion que el movimiento sea válido y con el Emisor notifica al tablero de su movimiento.

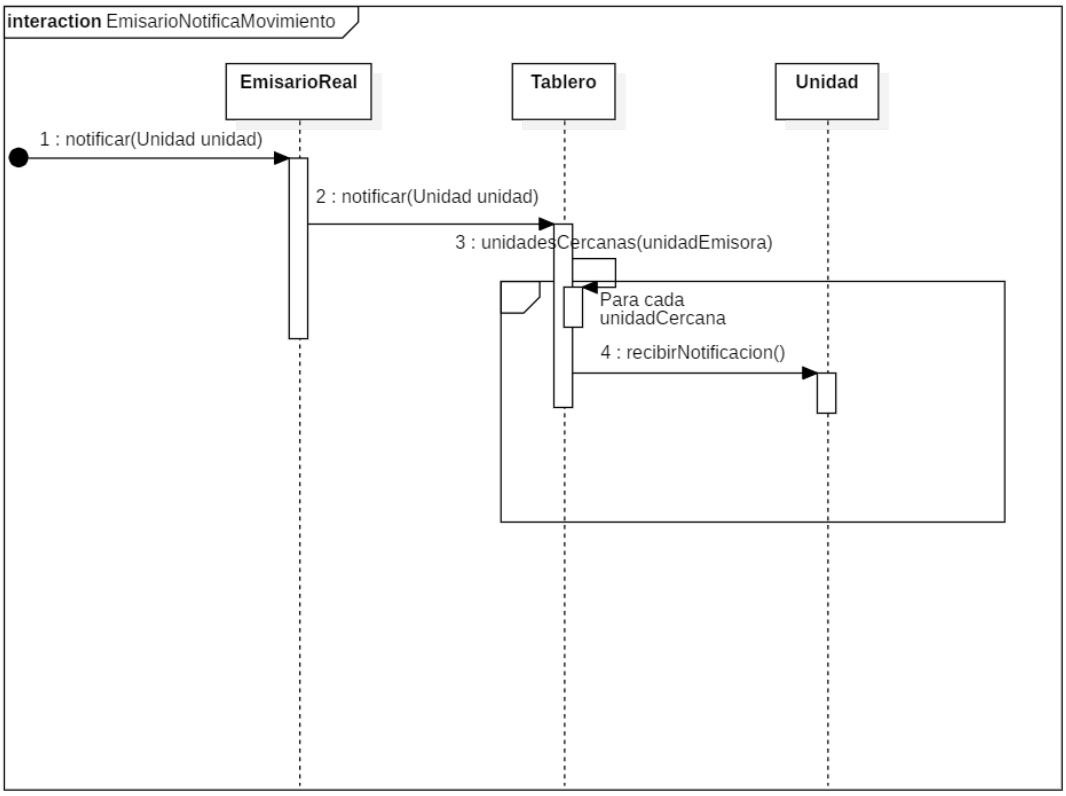

Figura 7: Diagrama de Secuencia caso EmisarioReal.

Por la aplicación del null pattern el Emisario puede ser real.º "nulo", como estamos moviendo un soldado con EmisarioReal hacia las cercanias de un Jinete, este es real", el mismo notifica el movimiento de la unidad al tablero, el cual busca las unidades cercanas a la unidad emisora y para cada una de ellas ejecuta el método de la clase Unidad recibirNotificacion()".

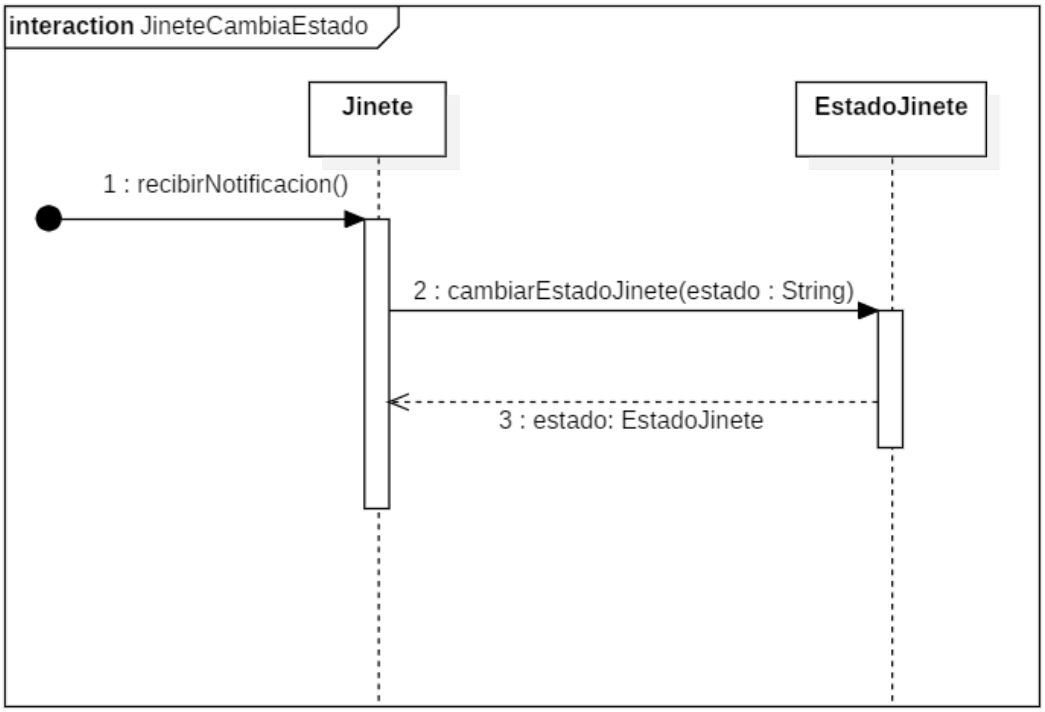

Figura 8: Diagrama de Secuencia caso unidad es Jinete.

La Unidad, en este caso, Jinete, recibe la notificacion y como ahora, hay un soldado cercano, se le envía a la interface EstadoJinete el mensaje çambiarEstadoJinete() que devuelve el nuevo estado del Jinete, para entender de una mejor forma la interface veamos la Figura 9

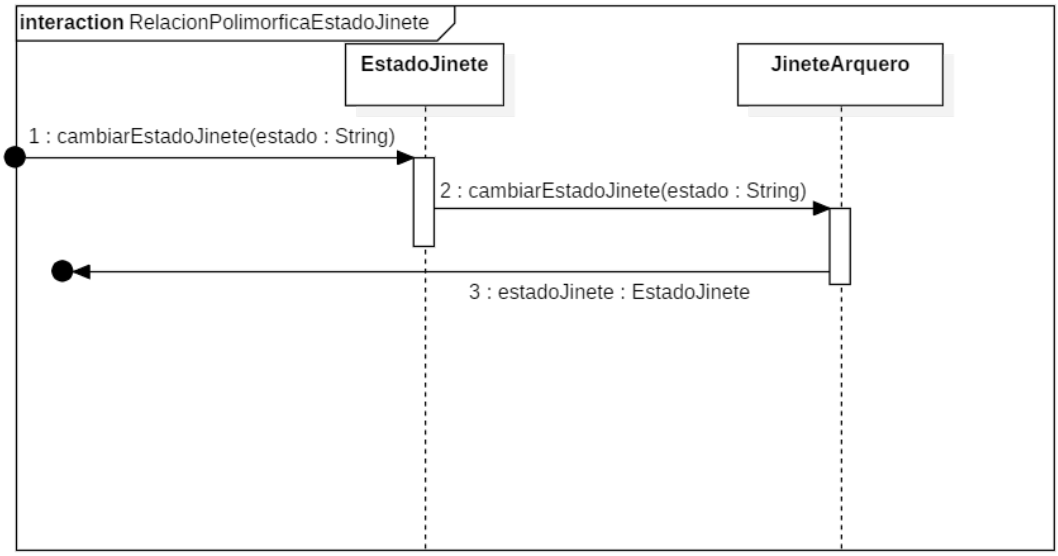

Figura 9: Diagrama de Secuencia Relacion polimorfica EstadoJinete.

Como el caso a analizar era un jinete que no tenía unidades enemigas cercanas, pero luego un soldado se movió a una posición próxima a él, el estado del jinete era un JineteArquero, entonces la interacción se dá desde EstadoJinete a JineteArquero, jineteArquero, hereda el método cambiarEstadoJinete, y devuelve una nueva instancia de EstadoJinete, en el caso de este problema va a ser una instancia de JineteEspadachín, ya que tiene un soldado enemigo cercano.