Algoritmos III
Bugliot, Nicolás
Karagoz, Filyan
Martinez, Santiago
Ruiz Huidobro, Matías

Algo-Chess

TP3

Introducción:

El objetivo de este trabajo práctico es el desarrollo grupal de una aplicación, aplicando los conceptos aprendidos durante el curso. Para la implementación se utilizará Java, con un diseño del modelo orientado a objetos y trabajando con las técnicas de TDD e Integración Continua. La aplicación será desarrollada en su totalidad siguiendo el modelo MVC (Modelo Vista Cotrolador), por lo que contará con una interfaz gráfica.

La aplicación consiste en un videojuego por turnos, de dos jugadores conformado de un tablero y distintas unidades sobre él. El objetivo del juego es destruir todas las unidades enemigas y gana el jugador que lo consigue primero. El jugador dispone de distintas entidades para acomodar en el tablero, pero tiene una cantidad limitada de puntos para usar (cada unidad cuenta con un costo distinto).

Supuestos:

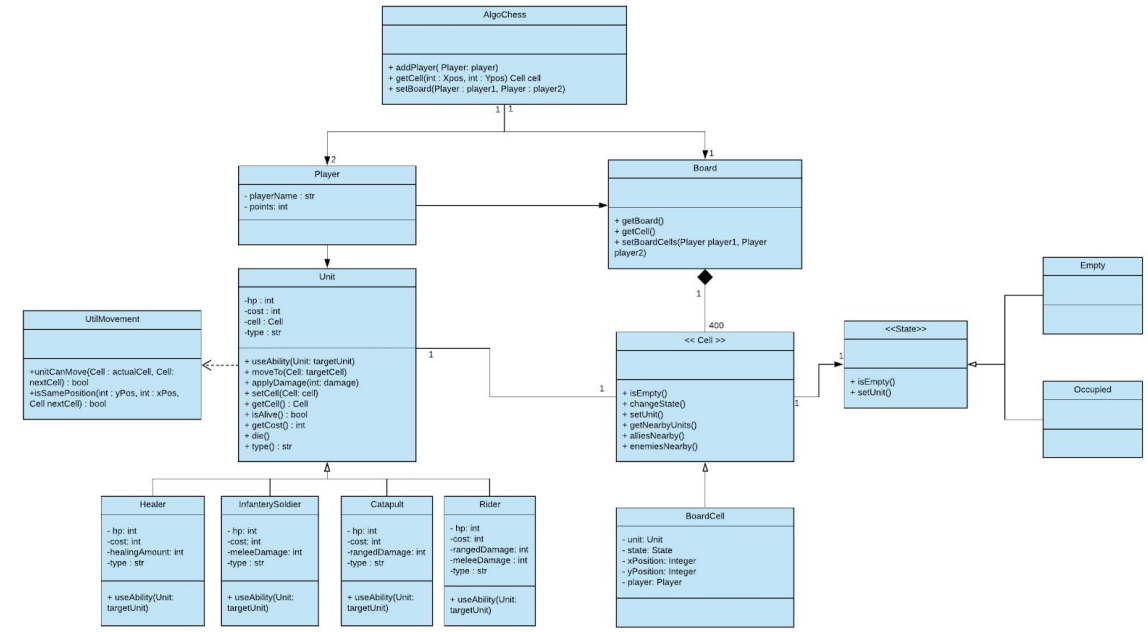
Los supuestos que tomamos para este trabajo práctico fueron:

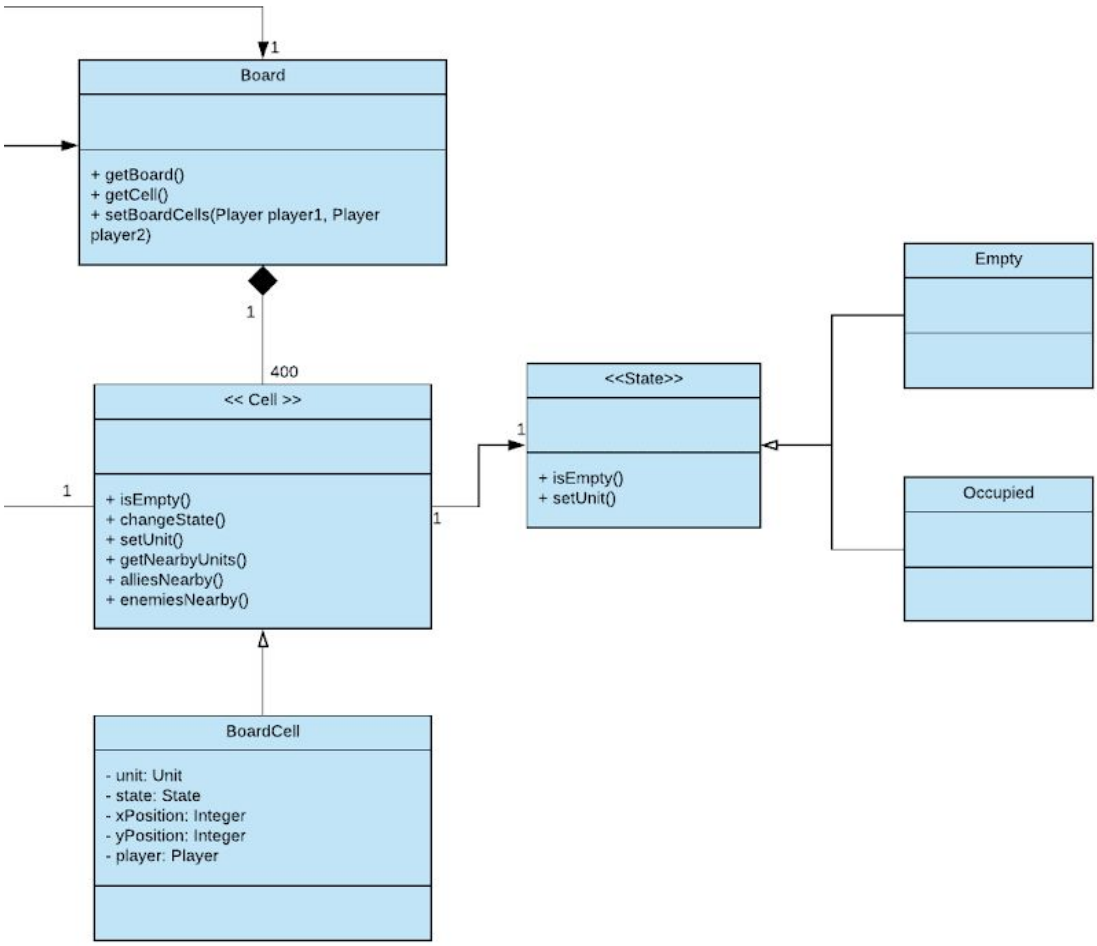
- Permite hasta dos jugadores.
- Los movimientos se pueden dar en dirección horizontal, vertical o diagonal.
- Dos unidades o más que forman una diagonal se consideran contiguas.

Diagramas de Clases:

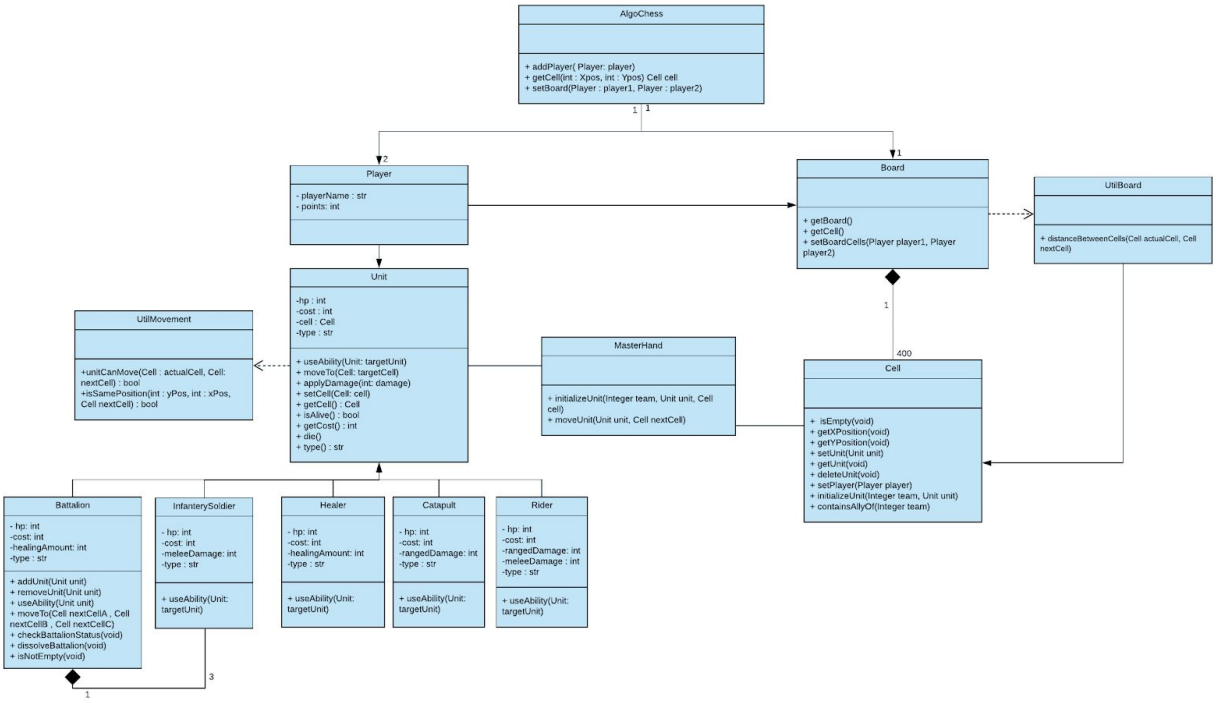
Primer Entrega:

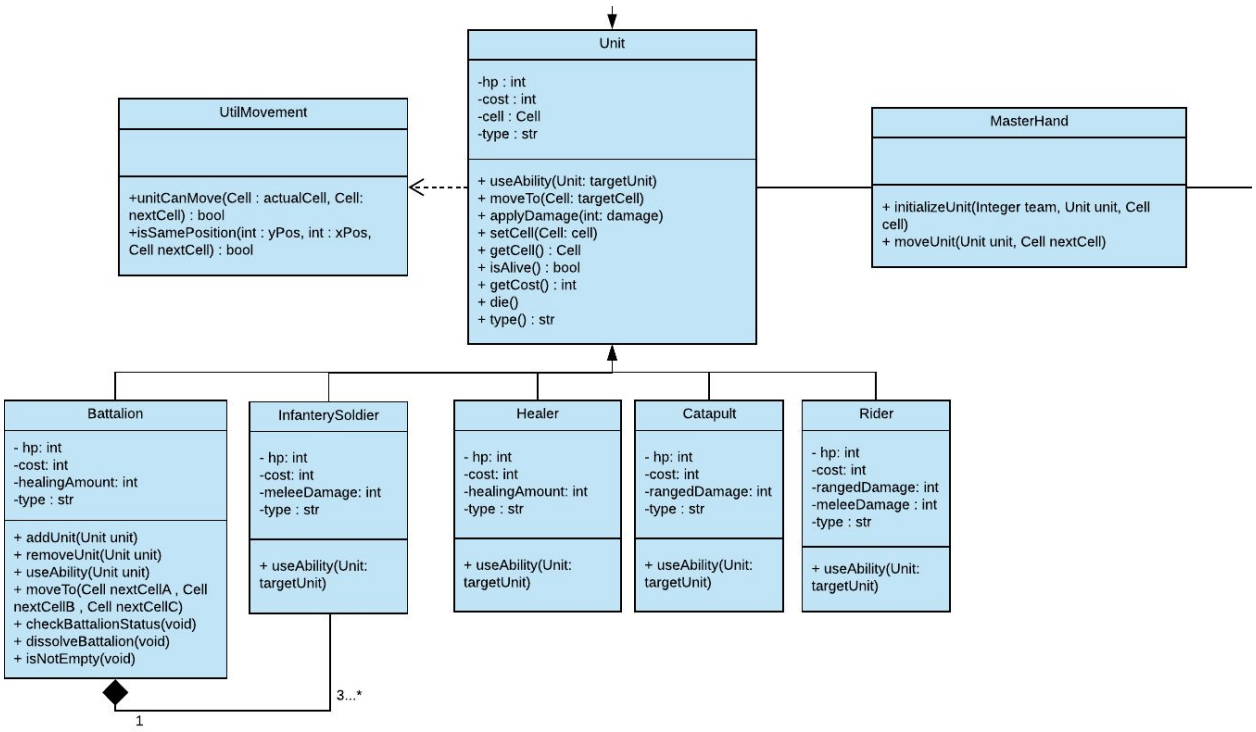
Para esta primer entrega se conceptualizaron los siguientes diagramas de clases como un primer boceto de las clases necesarias para lograr las interacciones requeridas.

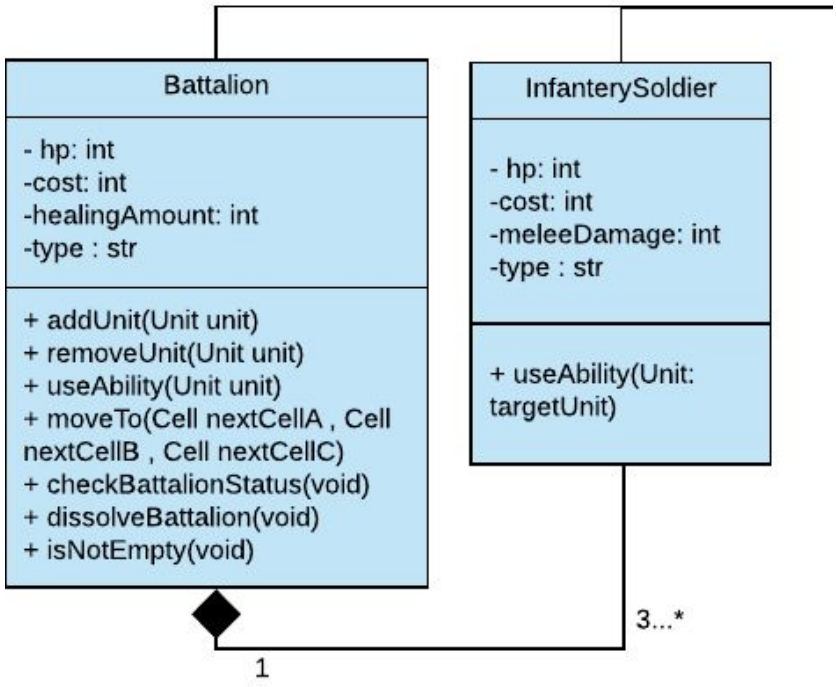

AlgoChess
+ addPlayer(Player: player)
+ getCell(int : Xpos, int : Ypos) Cell cell
+ setBoard(Player : player1, Player : player2)
1
1
2
1
Player
- playerName : str
- points: int
Board
+ getBoard()
+ getCell()
+ setBoardCells(Player player1, Player player2)
Unit
-hp : int
-cost : int
-cell : Cell
-type : str
+ useAbility(Unit: targetUnit)
+ moveTo(Cell: targetCell)
+ applyDamage(int: damage)
+ setCell(Cell: cell)
+ getCell() : Cell
+ isAlive() : bool
+ getCost() : int
+ die()
+ type() : str
UtilMovement
+unitCanMove(Cell : actualCell, Cell: nextCell) : bool
+isSamePosition(int : yPos, int : xPos, Cell nextCell) : bool
1
400
<< Cell >>
+ isEmpty()
+ changeState()
+ setUnit()
+ getNearbyUnits()
+ alliesNearby()
+ enemiesNearby()
1
1
<<State>>
+ isEmpty()
+ setUnit()
1
Empty
Occupied
Healer
- hp: int
-cost: int
-healingAmount: int
-type : str
+ useAbility(Unit: targetUnit)
InfanterySoldier
- hp: int
-cost: int
-meleeDamage: int
-type : str
+ useAbility(Unit: targetUnit)
Catapult
- hp: int
-cost: int
-rangedDamage: int
-type : str
+ useAbility(Unit: targetUnit)
Rider
- hp: int
-cost: int
-rangedDamage: int
-meleeDamage : int
-type : str
+ useAbility(Unit: targetUnit)
BoardCell
- unit: Unit
- state: State
- xPosition: Integer
- yPosition: Integer
- player: Player

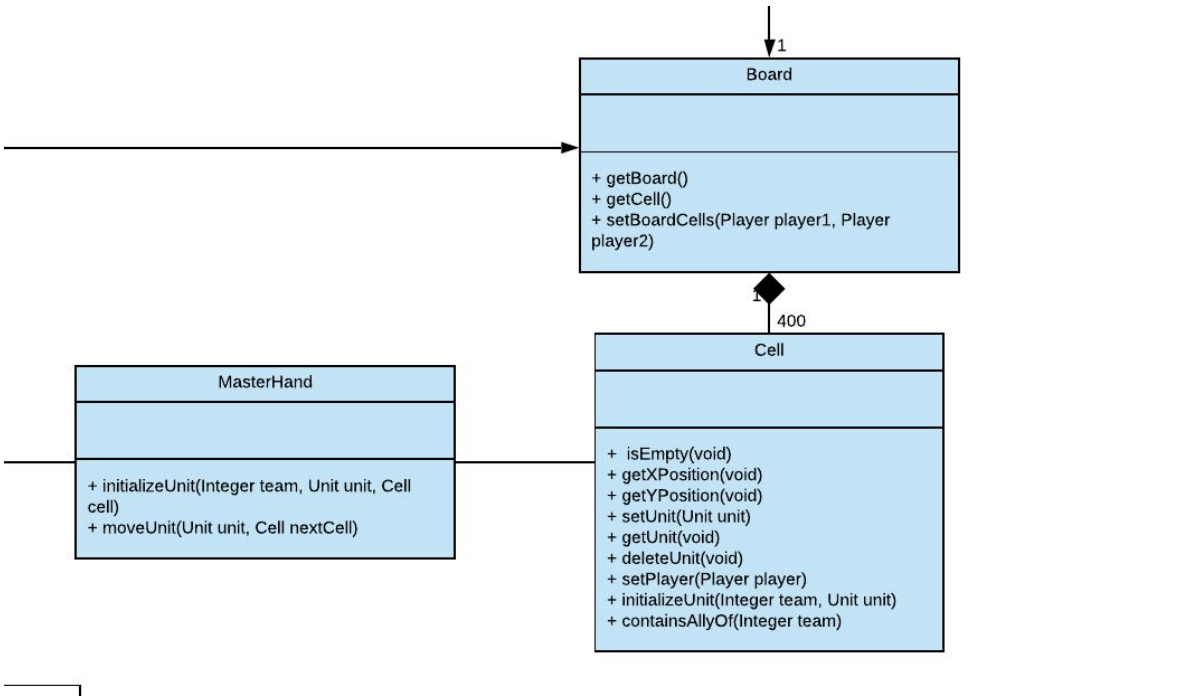

1
Board
+ getBoard()
+ getCell()
+ setBoardCells(Player player1, Player player2)
1
400
<< Cell >>
+ isEmpty()
+ changeState()
+ setUnit()
+ getNearbyUnits()
+ alliesNearby()
+ enemiesNearby()
1
1
<<State>>
+ isEmpty()
+ setUnit()
1
Empty
Occupied
BoardCell
- unit: Unit
- state: State
- xPosition: Integer
- yPosition: Integer
- player: Player

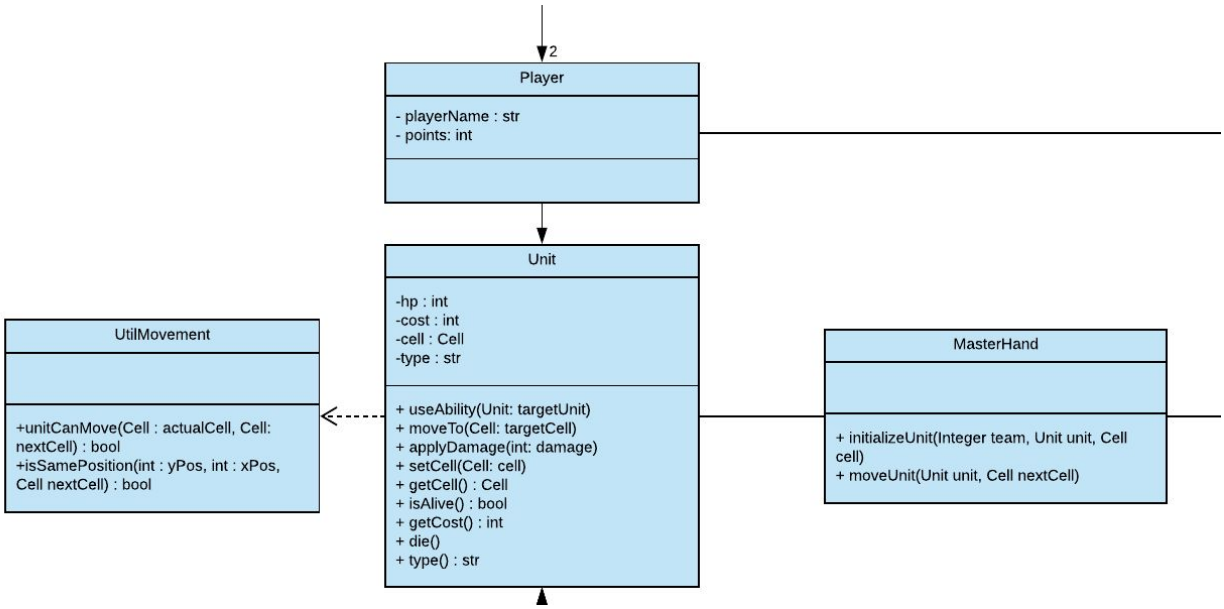
Segunda Entrega:




Unit
-hp : int
-cost : int
-cell : Cell
-type : str
+ useAbility(Unit: targetUnit)
+ moveTo(Cell: targetCell)
+ applyDamage(int: damage)
+ setCell(Cell: cell)
+ getCell() : Cell
+ isAlive() : bool
+ getCost() : int
+ die()
+ type() : str
UtilMovement
+unitCanMove(Cell : actualCell, Cell: nextCell) : bool
+isSamePosition(int : yPos, int : xPos, Cell nextCell) : bool
MasterHand
+ initializeUnit(Integer team, Unit unit, Cell cell)
+ moveUnit(Unit unit, Cell nextCell)
Battalion
- hp: int
-cost: int
-healingAmount: int
-type : str
+ addUnit(Unit unit)
+ removeUnit(Unit unit)
+ useAbility(Unit unit)
+ moveTo(Cell nextCellA , Cell nextCellB , Cell nextCellC)
+ checkBattalionStatus(void)
+ dissolveBattalion(void)
+ isNotEmpty(void)
InfanterySoldier
- hp: int
-cost: int
-meleeDamage: int
-type : str
+ useAbility(Unit: targetUnit)
Healer
- hp: int
-cost: int
-healingAmount: int
-type : str
+ useAbility(Unit: targetUnit)
Catapult
- hp: int
-cost: int
-rangedDamage: int
-type : str
+ useAbility(Unit: targetUnit)
Rider
- hp: int
-cost: int
-rangedDamage: int
-meleeDamage : int
-type : str
+ useAbility(Unit: targetUnit)
3...*
1


Battalion
- hp: int
-cost: int
-healingAmount: int
-type : str
+ addUnit(Unit unit)
+ removeUnit(Unit unit)
+ useAbility(Unit unit)
+ moveTo(Cell nextCellA , Cell nextCellB , Cell nextCellC)
+ checkBattalionStatus(void)
+ dissolveBattalion(void)
+ isNotEmpty(void)
InfanterySoldier
- hp: int
-cost: int
-meleeDamage: int
-type : str
+ useAbility(Unit: targetUnit)
3...*
1

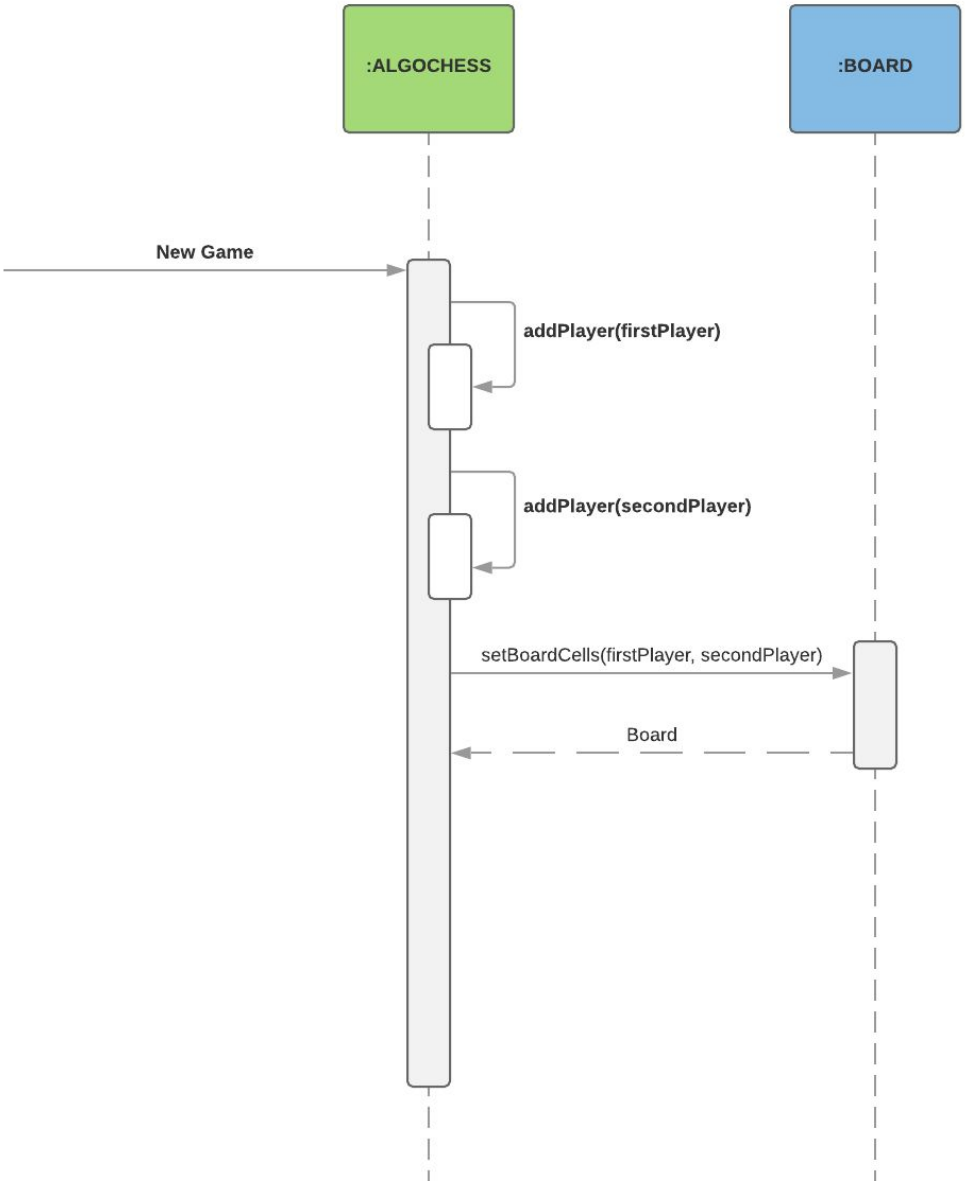

1
Board
+ getBoard()
+ getCell()
+ setBoardCells(Player player1, Player player2)
1
400
Cell
+ isEmpty(void)
+ getXPosition(void)
+ getYPosition(void)
+ setUnit(Unit unit)
+ getUnit(void)
+ deleteUnit(void)
+ setPlayer(Player player)
+ initializeUnit(Integer team, Unit unit)
+ containsAllyOf(Integer team)
MasterHand
+ initializeUnit(Integer team, Unit unit, Cell cell)
+ moveUnit(Unit unit, Cell nextCell)


2
Player
- playerName : str
- points: int
Unit
-hp : int
-cost : int
-cell : Cell
-type : str
+ useAbility(Unit: targetUnit)
+ moveTo(Cell: targetCell)
+ applyDamage(int: damage)
+ setCell(Cell: cell)
+ getCell() : Cell
+ isAlive() : bool
+ getCost() : int
+ die()
+ type() : str
UtilMovement
+unitCanMove(Cell : actualCell, Cell: nextCell) : bool
+isSamePosition(int : yPos, int : xPos, Cell nextCell) : bool
MasterHand
+ initializeUnit(Integer team, Unit unit, Cell cell)
+ moveUnit(Unit unit, Cell nextCell)

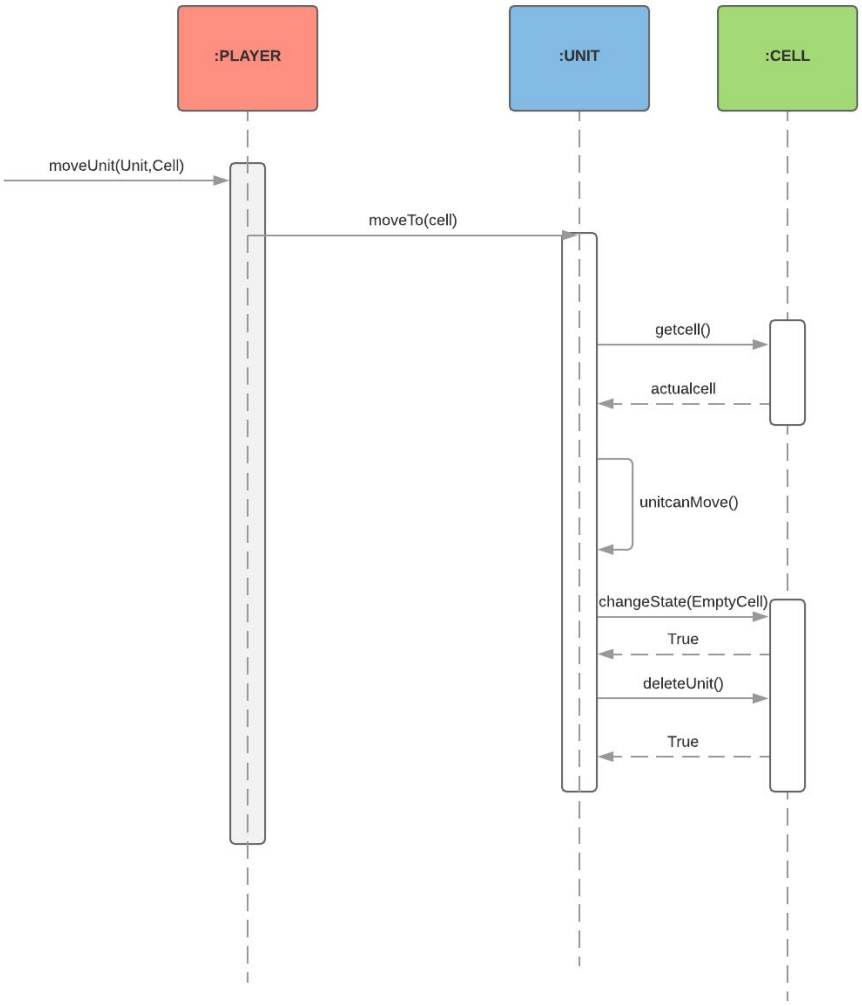
Diagramas de Secuencia:

Primera Entrega:

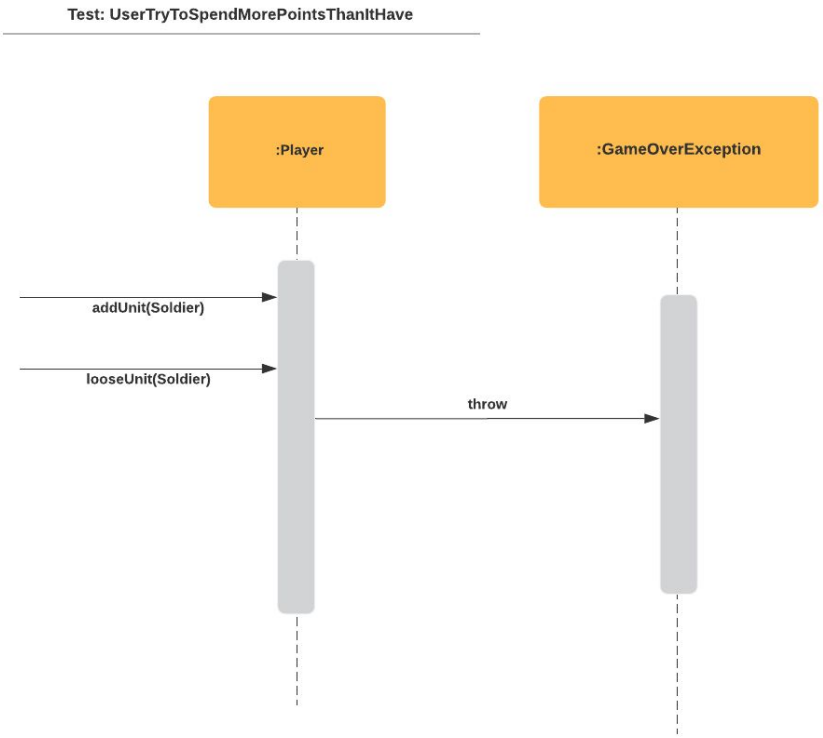
Creación del tablero

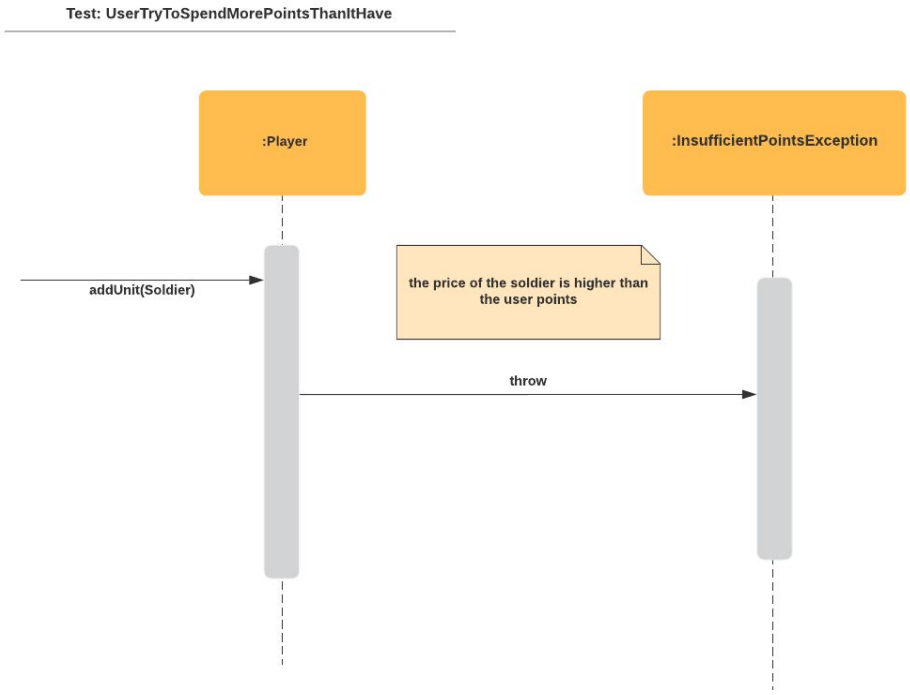


Mover una pieza

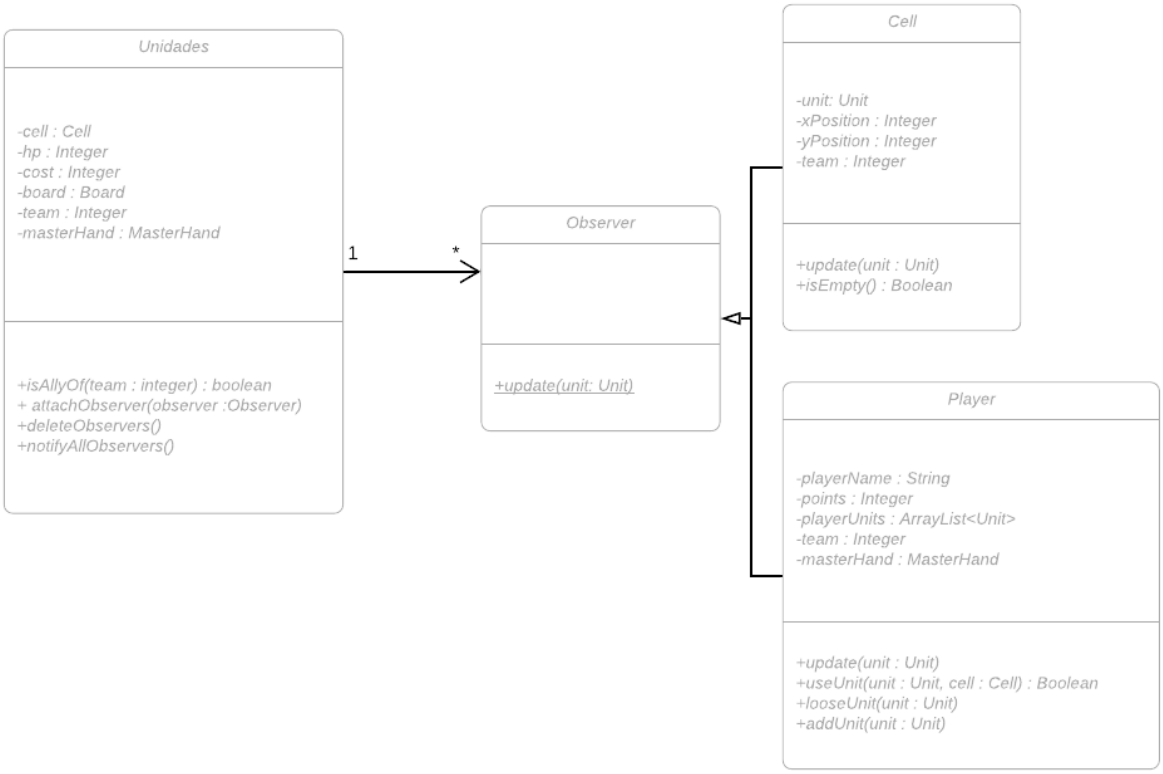


Segunda Entrega:

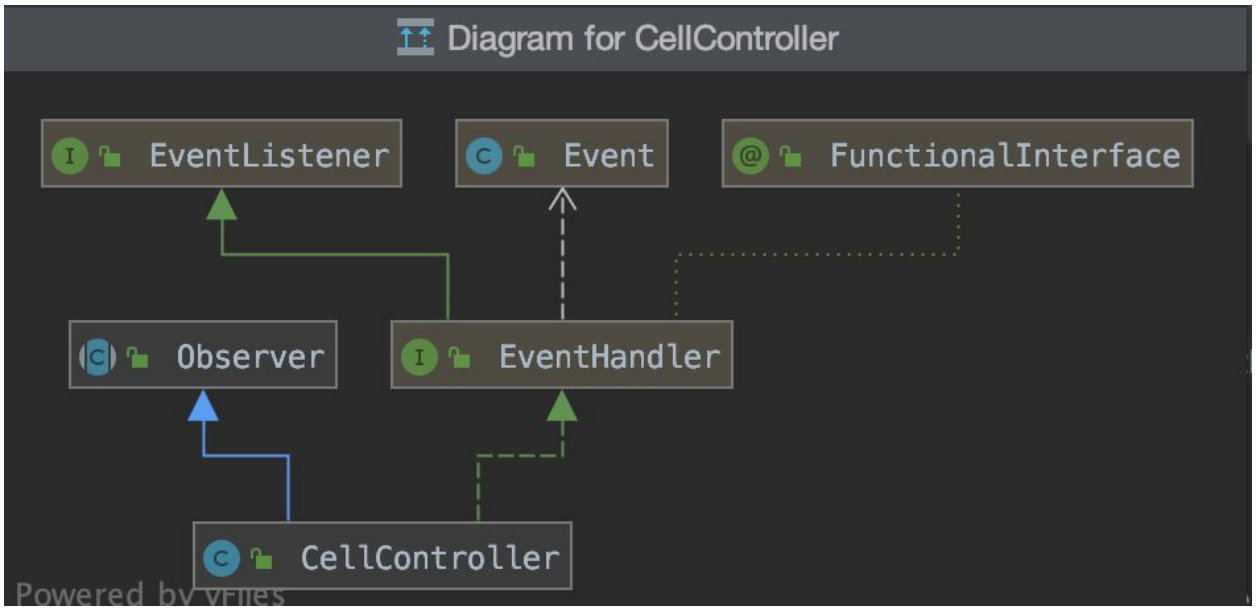


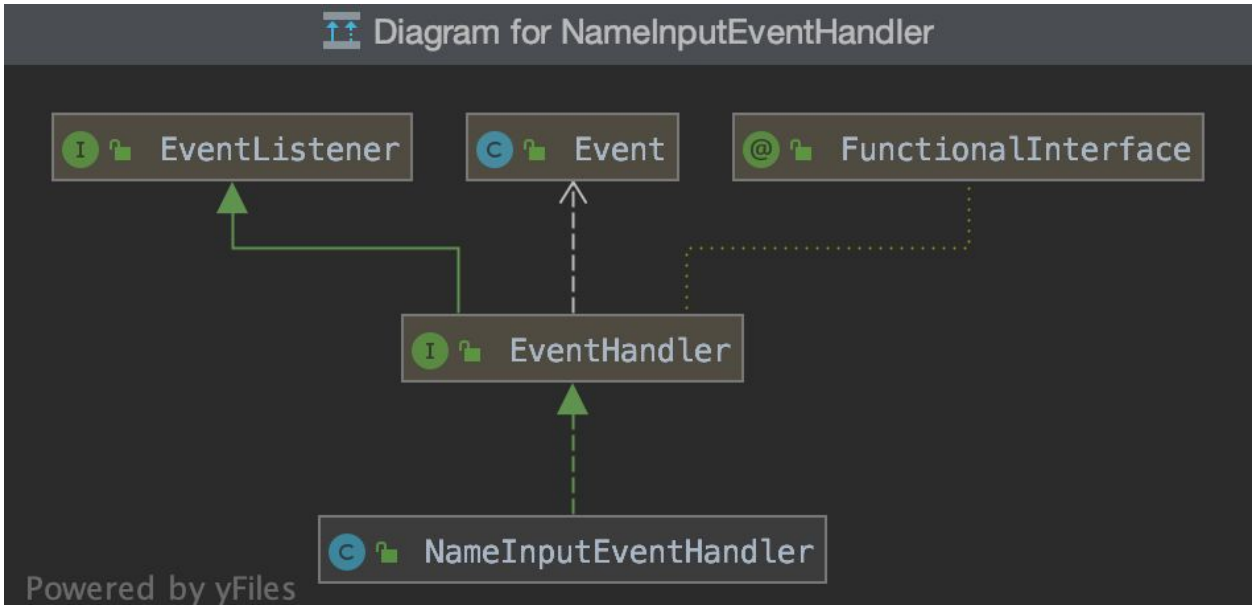


Cuarta Entrega:


Unidades
-cell : Cell
-hp : Integer
-cost : Integer
-board : Board
-team : Integer
-masterHand : MasterHand
+isAllyOf(team : integer) : boolean
+ attachObserver(observer :Observer)
+deleteObservers()
+notifyAllObservers()
1
*
Observer
+update(unit: Unit)
Cell
-unit: Unit
-xPosition : Integer
-yPosition : Integer
-team : Integer
+update(unit : Unit)
+isEmpty() : Boolean
Player
-playerName : String
-points : Integer
-playerUnits : ArrayList<Unit>
-team : Integer
-masterHand : MasterHand
+update(unit : Unit)
+useUnit(unit : Unit, cell : Cell) : Boolean
+looseUnit(unit : Unit)
+addUnit(unit : Unit)

Diagramas de Controladores:





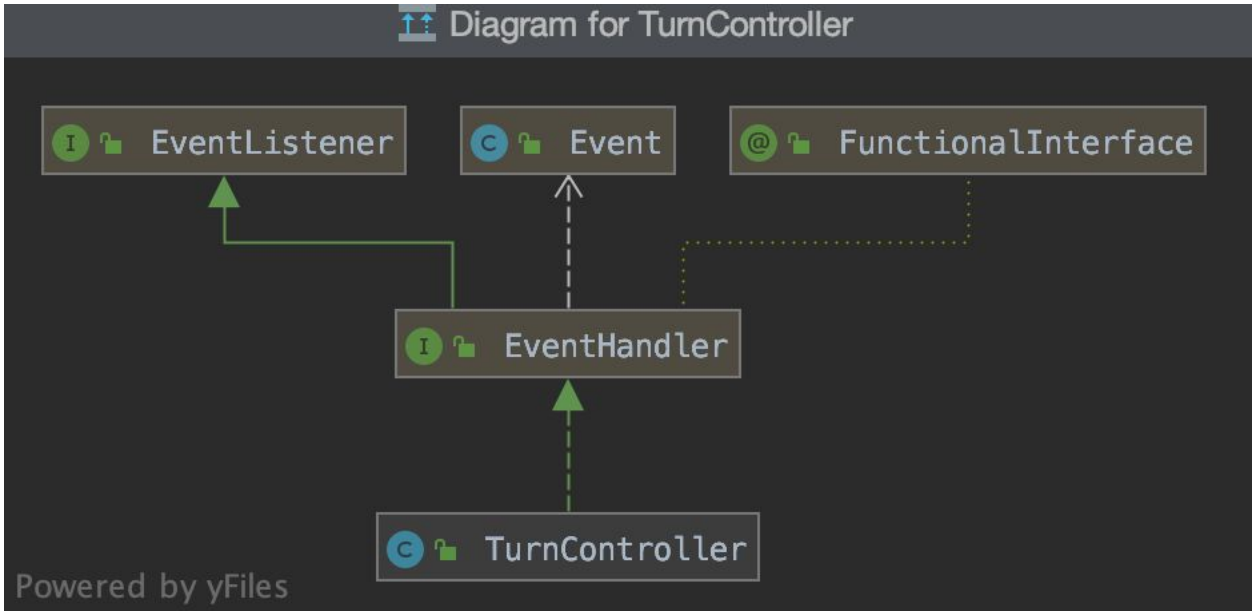


Diagram for TurnController
EventListener
Event
FunctionalInterface
EventHandler
TurnController
Powered by yFiles

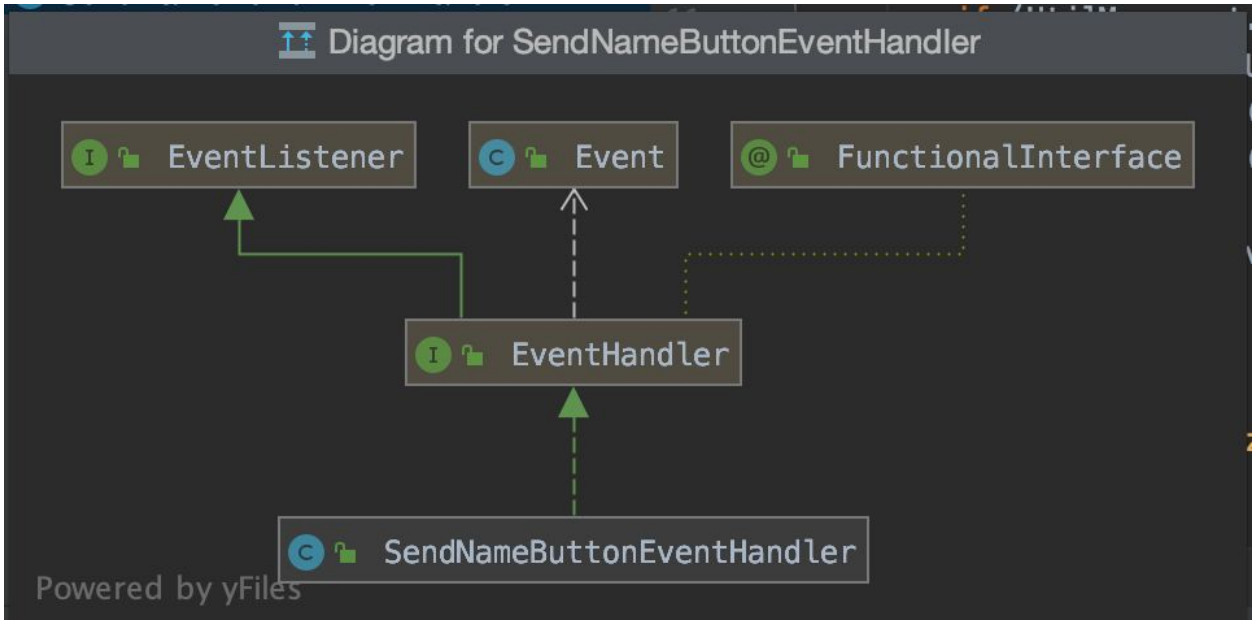
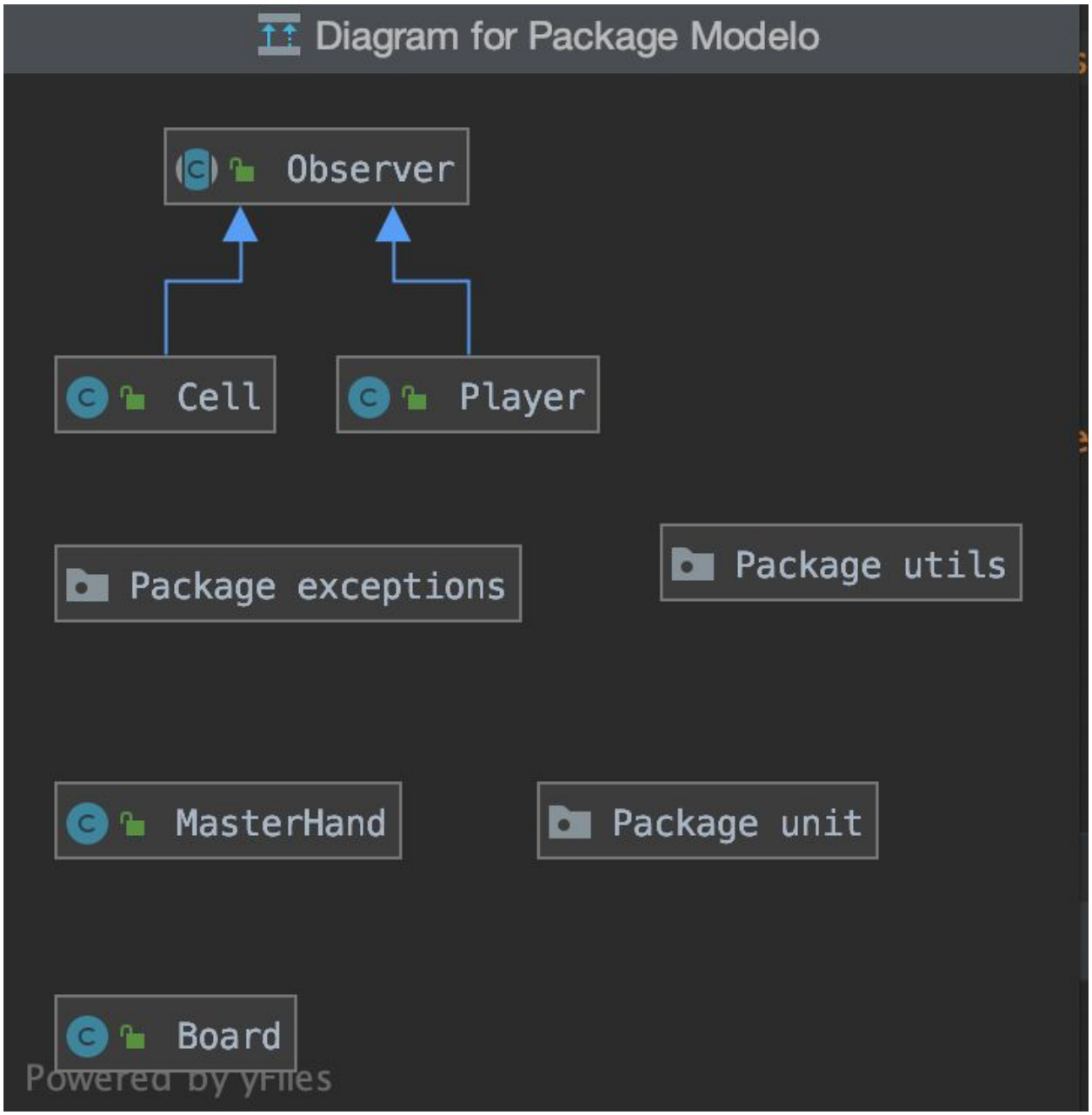


Diagram for SendNameButtonEventHandler
EventListener
Event
FunctionalInterface
EventHandler
SendNameButtonEventHandler
Powered by yFiles

Diagrama de Paquete Modelo:



Detalles de Implementación:

Primer Entrega:

Para esta primer entrega se planteó una estructura que intente imitar el diagrama de clases indicado previamente. Optamos por hacer una clase de Unidad (Unit) abstracta la cual concentraba el comportamiento general a todas las unidades y polimórficamente tratar las interacciones entre unidades. Esto se realizó optando por qué en vez de una unidad atacar o curar, cada unidad tenga una o varias habilidades. Entonces a la unidad se le indicará que utilice su habilidad (método UseAbility) sobre otra, y luego en las implementaciones hijas de Unidad, se defina como comportarse ante este método.

También creamos una unidad extra para las pruebas, un “Muñeco de Prueba” (TestDummy) el cual nos permitió realizar pruebas se ataques sobre él, obteniendo valores específicos de vida y daño, sin comprometer el encapsulamiento de las otras unidades.

Segunda Entrega:

En esta fase, requerimos realizar pruebas en las que varias clases interactúan entre sí. Es uno de estos casos el del jinete, para el cual tuvimos que implementar una técnica de Mocking para simular la interacción con el tablero de manera controlada. Esto nos permitió tanto trabajar en las clases de tablero y unidades a la vez, ya que no requerimos la terminación del tablero para poder verificar el funcionamiento de las unidades, y también la corroboración de que los métodos específicos de ataque del jinete estaban funcionando de manera esperada, ya que el comportamiento de las otras clases era simulado y controlado.

También utilizamos una técnica de Espionaje (Spy) para simular el comportamiento específico de uno de los métodos de las unidades con la que interactúan. De esta forma al preguntarle a una unidad a qué distancia se encontraba, podíamos controlar el resultado de ese método, pero mantener el resto del comportamiento normal de la unidad.

Cuarta Entrega:

Empleamos el patrón Singleton para el Tablero, ya que sería necesaria sólo una instancia de la clase, y varias clases iban a tener que interactuar con esta. Por lo tanto optamos por darle un acceso global al Tablero.

Durante el transcurso del juego, los principales actores iban a ser las unidades, quienes iban a estar cambiando constantemente su estado interno. Las vistas iban a precisar conocer el estado de las unidades y sus cambios, para poder correctamente representarlas. Es por esto que optamos por implementar el patrón de diseño Observer sobre las mismas.

Excepciones:

Primer Entrega:

Las siguientes excepciones fueron creadas con el fin de obtener mayor información en caso de error y poder también abstraer los errores de las implementaciones propias de Java.

Excepción de Celda Vacía: **EmptyCellException**

Excepción de Celda Ocupada: **OccupiedCellException**

Excepción de Fin De Juego: **GameOverException**

Excepción de Puntos Insuficientes: **InsufficientPointsException**

Excepción de Movimiento: **MovementException**

Fueron creadas de Manera Preemptiva para disponer de ellas a futuro durante la siguiente fase de desarrollo, donde más entidades estarán interactuando entre sí y consideramos nos serán útiles.

Segunda Entrega:

Las previamente mencionadas excepciones fueron adaptadas, utilizandolas como casos de error dentro de las clases particulares a las que hacen referencia sus nombres. Estas excepciones dan un mensaje particular dependiendo del caso en el que fueron lanzadas, pero su nombre general hace referencia a la clase de la que provienen. De este modo podemos prevenir y adaptarnos mejor a los posibles errores que desencadenan en excepciones en tiempo de ejecución, previniendo la terminación del programa.

Cuarta Entrega:

Se implementó la interfaz gráfica, y en ella se aprovecharon las excepciones utilizando una vista del estilo PopUp genérica, que podía ser construida utilizando el mensaje de error que daba la excepción. De esta manera las excepciones creadas fueron utilizadas para indicarle al jugador cuando realizando acciones que rompían las “reglas” del juego, sirviendo tanto para enseñarle a jugar como para dar una explicación de porque eso que el usuario intento no funcionó como esperaba.