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:

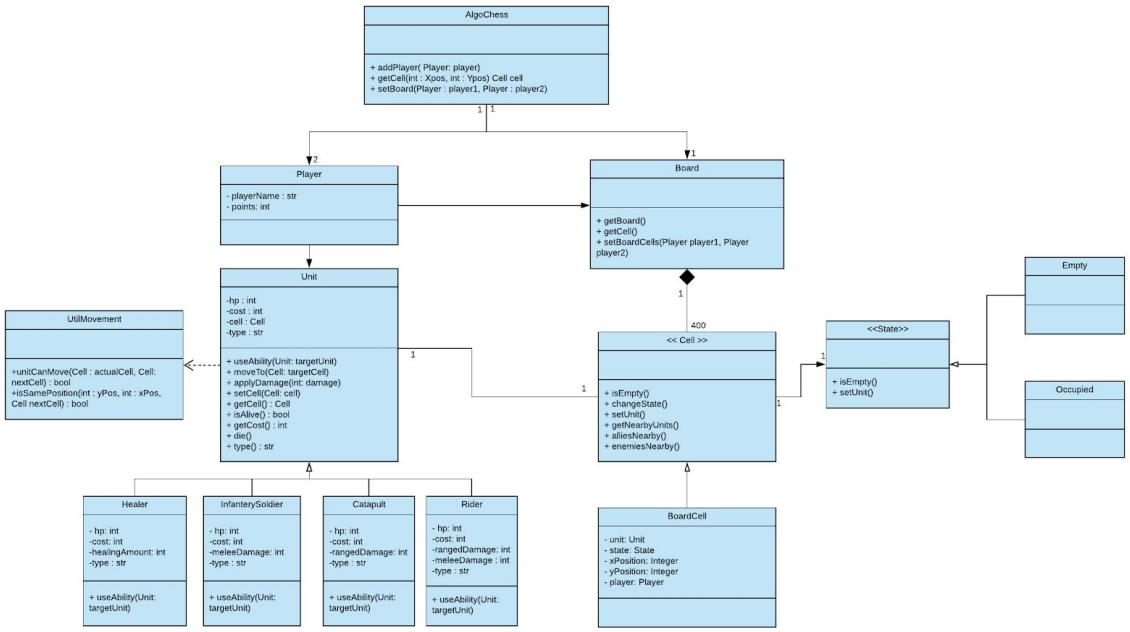
Los supuestos que tomamos para este trabajo práctico fueron:

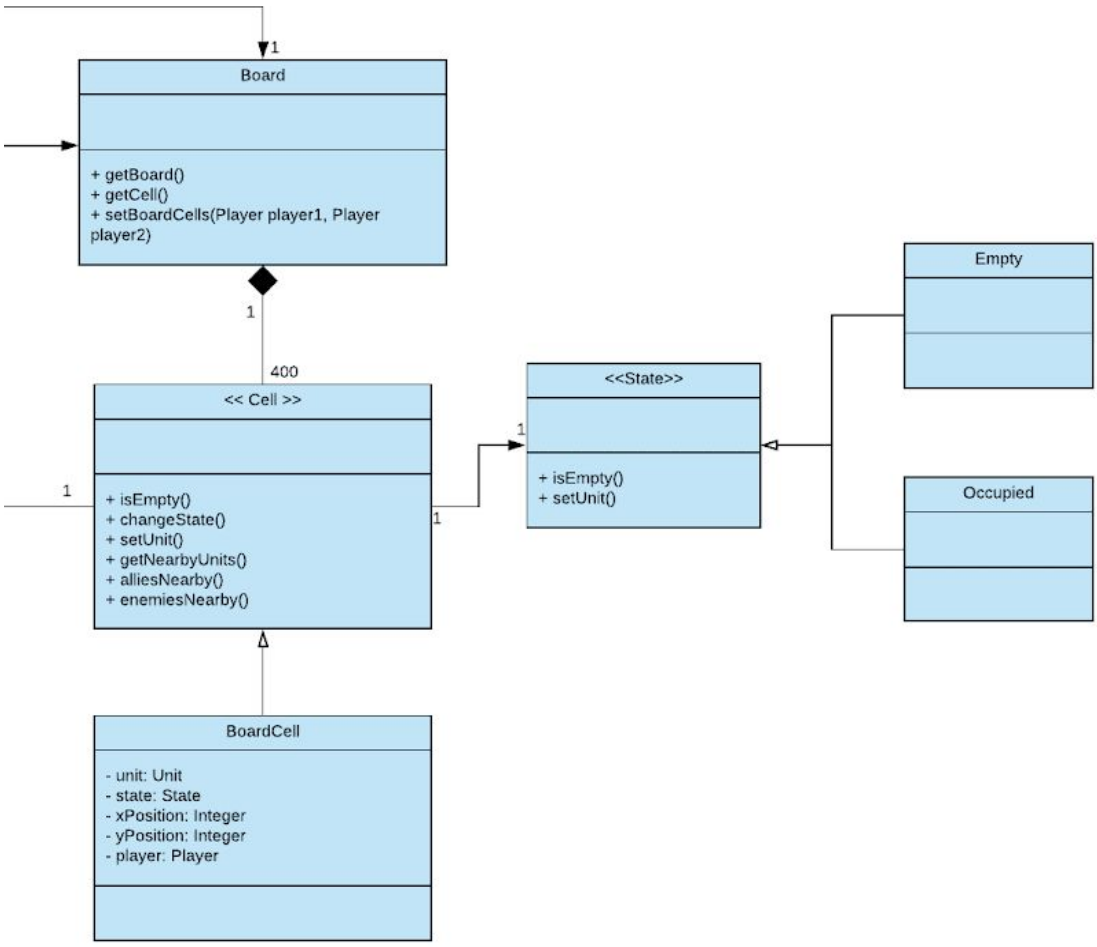
- Permite hasta dos jugadores.

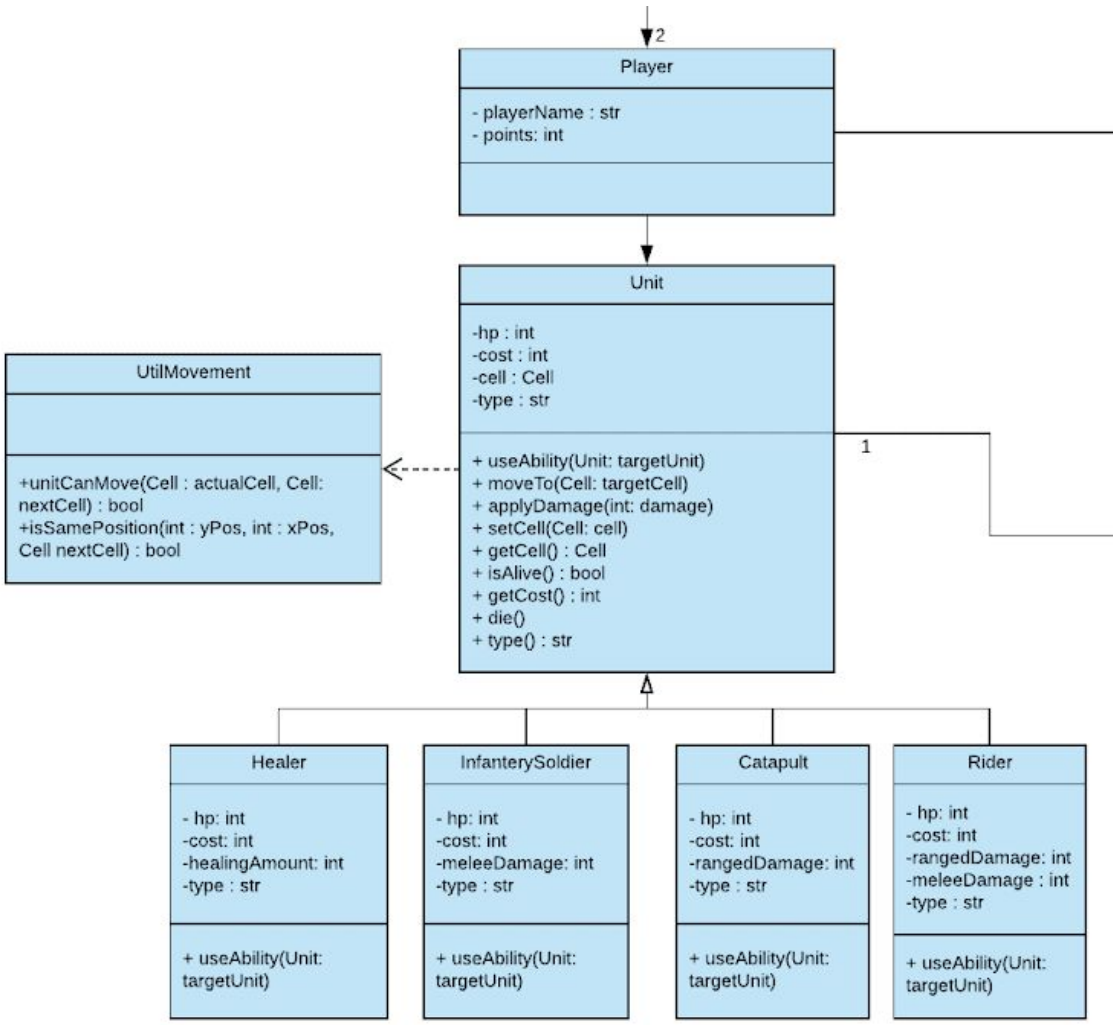
Diagramas de Clases:

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.

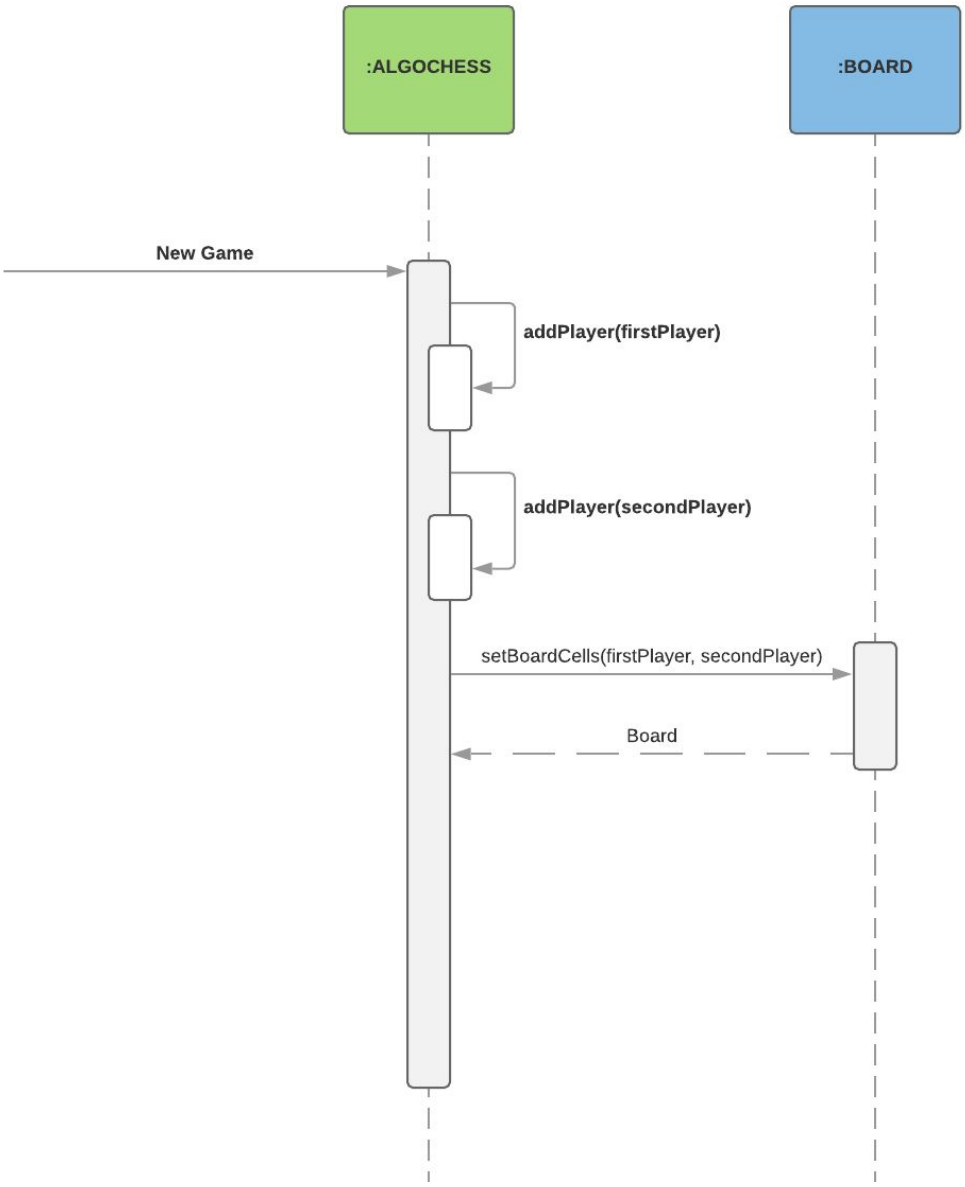

AlgoChess
+ addPlayer(Player: player)
+ getCell(int : Xpos, int : Ypos) Cell cell
+ setBoard(Player : player1, Player : player2)
1
1
2
1
Player
- playerName : str
- points: int
Board
+ getBoard()
+ getCell()
+ setBoardCells(Player player1, Player player2)
Unit
-hp : int
-cost : int
-cell : Cell
-type : 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
1
400
<< Cell >>
+ isEmpty()
+ changeState()
+ setUnit()
+ getNearbyUnits()
+ alliesNearby()
+ enemiesNearby()
1
1
<<State>>
+ isEmpty()
+ setUnit()
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 >>
1
+ isEmpty()
+ changeState()
+ setUnit()
+ getNearbyUnits()
+ alliesNearby()
+ enemiesNearby()
1
<<State>>
1
+ isEmpty()
+ setUnit()
Empty
Occupied
BoardCell
- unit: Unit
- state: State
- xPosition: Integer
- yPosition: Integer
- player: Player

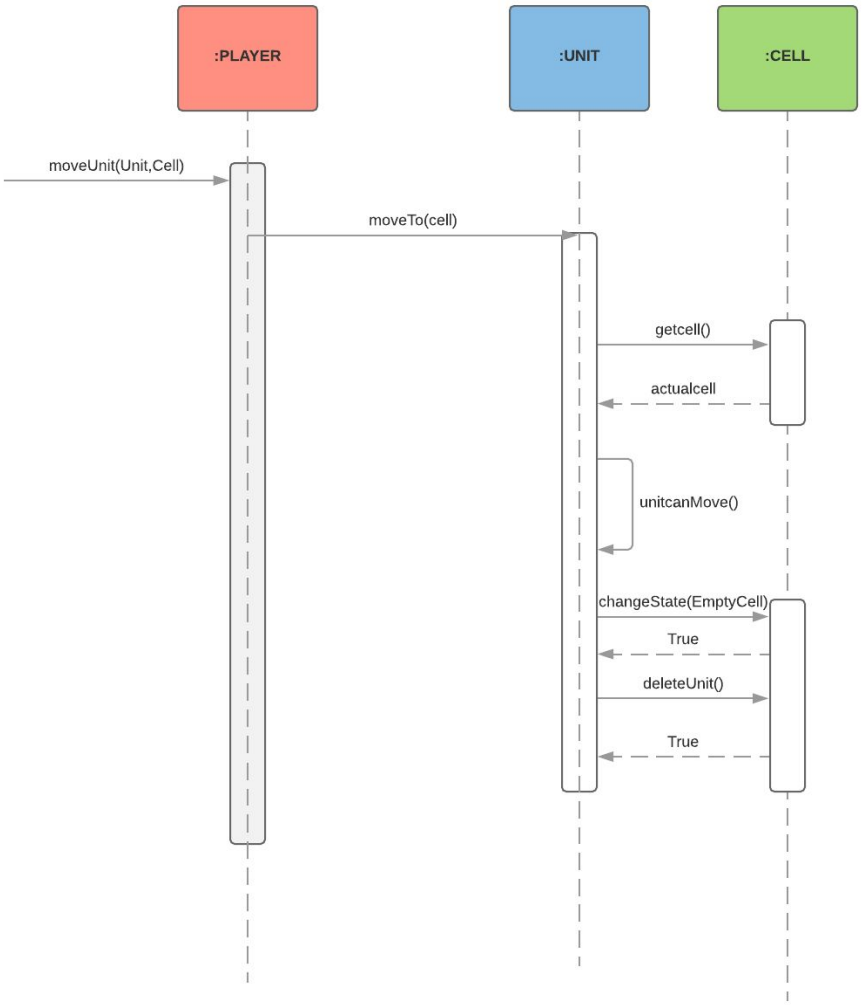

2
Player
- playerName : str
- points: int
Unit
-hp : int
-cost : int
-cell : Cell
-type : str
+ useAbility(Unit: targetUnit)
+ moveTo(Cell: targetCell)
+ applyDamage(int: damage)
+ setCell(Cell: cell)
+ getCell() : Cell
+ isAlive() : bool
+ getCost() : int
+ die()
+ type() : str
1
UtilMovement
+unitCanMove(Cell : actualCell, Cell: nextCell) : bool
+isSamePosition(int : yPos, int : xPos, Cell nextCell) : bool
Healer
- hp: int
-cost: int
-healingAmount: int
-type : str
+ useAbility(Unit: targetUnit)
InfantrySoldier
- 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)

Diagramas de Secuencia:

Creación del tablero



Mover una pieza



Diagramas de Paquetes:

Diagramas de Estado:

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.

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.