Assignment 2 – Group 16

Exercise 1 – Getters and Setters out

Due to the refactoring of our code, many setters and getters were no longer necessary. In other cases, we directly replaced them with different functions. Below are nine

1. Board: replace getBoard() with functions fieldIsEmpty(), fieldContainsKing(), fieldContainsCheckerColor().
2. Player: remove getCheckers(), setCanMove(), getCheckerPosition(). Now we only need findPlayerCheckers() thanks to some refactoring. We moved this function to the Game class.
3. Player: Removed getColor() and replaced with isRedPlayersTurn(), also moved to Game class.
4. Game: isValidMove() variable and getter/setter not used anymore (refactoring).
5. Game: setAllPossibleMoves() and getAllPossibleMoves() removed (refactoring).
6. Checker: Removed setPossibleMoves(), getPossibleMoves() as Arraylist is now stored in the Game class (refactoring).
7. Checker: removed get and set x&y coordinates as not needed due to refactoring.
8. Checker: removed getColor() and replaced with isWhitePlayerChecker() and String playerColor().
9. Move (formerly named Coordinate): removed getS() and replaced with isMoveJump().

The only getters we left are in the Move class. The main purpose of the Move class is to store a set of Integers, which is less confusing to deal with than e.g., a List of Integers. Since there is no way of changing the Integers after a Move object is initialized, we argue that the use of getters is appropriate here.

Exercise 2 – 20%-Time

Requirements

In addition to the requirements specified in the first assignment, we set ourselves two main goals:

1. Refactor the code to a greater degree than what was requested in exercise 1. For instance, we renamed the Coordinate class (now called Move), moved all the print statements and methods dealing with user input to the newly created UI class and improved the logic in the Game class by revising most methods and adding a few more, where necessary. We also realized that we did not need the Player class anymore and moved a method that found all the players checkers on the board to the Game class. (Anything else?)
2. We defined new requirements:
   When starting the program, the user is prompted to select a **strategy** for both the red and white **player**. There are three player strategies to choose from:
   Human: User is prompted to input their **move** on the **terminal**.
   Random: Dumb AI that selects a random (but valid) **move** from a list of possible moves.
   AI: Smarter AI that employs a simple **MinMax** algorithm to determine the next **move**, must be able to win against the random **strategy**.

Exercise 3 – Design Pattern

1. We decided to go with a strategy pattern. The strategy pattern presented itself as the best alternative because the Player class would have done something specific in a lot of different ways. In other words, it would have implemented the two Players either as human players or as an AI-bot (one with a random behavior and one using a Min-Max-Algorithm). The problem with this structure would have been that the class would have been hard to maintain because with every change the whole class would have been affected and the risk of introducing an error into the working code would have increased. In addition, it would not necessarily be easy to introduce new strategies without changing the context. With the strategy pattern all of these issues are being addressed.
   We implemented the strategy pattern by replacing the “Player” class in the model package with the context class (PlayerContext) and the interface (PlayerStrategy), over which the context interacts with the different strategy objects (in this case “HumanPlayer”, “MinMaxPlayer” and “RandomPlayer”). The different strategies can only be accessed via the interface “PlayerStrategy” is implemented by each strategy object.

2.

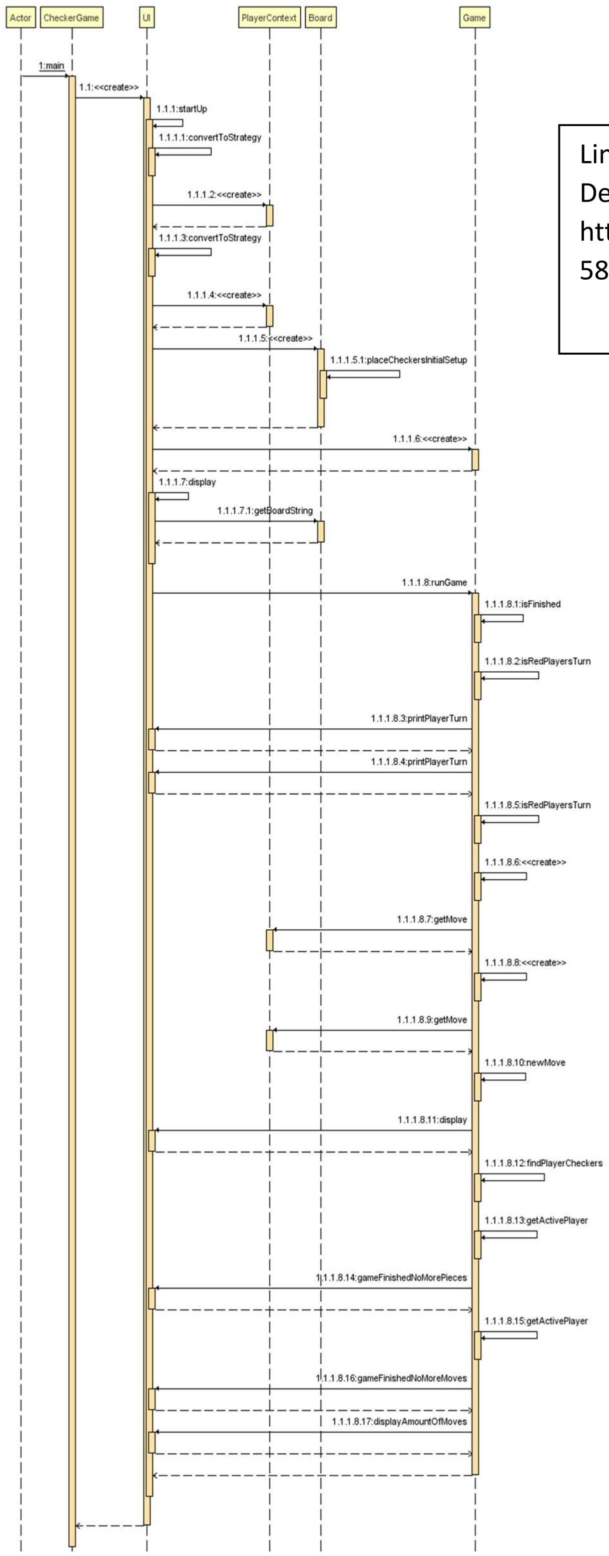
Actor
CheckerGame
UI
PlayerContext
Board
Game
1:main
1.1:<<create>>
1.1.1:startUp
1.1.1.1:convertToStrategy
1.1.1.2:<<create>>
1.1.1.3:convertToStrategy
1.1.1.4:<<create>>
1.1.1.5:<<create>>
1.1.1.5.1:placeCheckersInitialSetup
1.1.1.6:<<create>>
1.1.1.7:display
1.1.1.7.1:getBoardString
1.1.1.8:runGame
1.1.1.8.1:isFinished
1.1.1.8.2:isRedPlayersTurn
1.1.1.8.3:printPlayerTurn
1.1.1.8.4:printPlayerTurn
1.1.1.8.5:isRedPlayersTurn
1.1.1.8.6:<<create>>
1.1.1.8.7:getMove
1.1.1.8.8:<<create>>
1.1.1.8.9:getMove
1.1.1.8.10:newMove
1.1.1.8.11:display
1.1.1.8.12:findPlayerCheckers
1.1.1.8.13:getActivePlayer
1.1.1.8.14:gameFinishedNoMorePieces
1.1.1.8.15:getActivePlayer
1.1.1.8.16:gameFinishedNoMoreMoves
1.1.1.8.17:displayAmountOfMoves

Figure 1 Sequence Diagram (Call Depth 4)

Link to Sequence Diagram with a Call Depth of 6: https://www.swisstransfer.com/d/f5065589-1183-4705-8691-52eef2017eda

3.

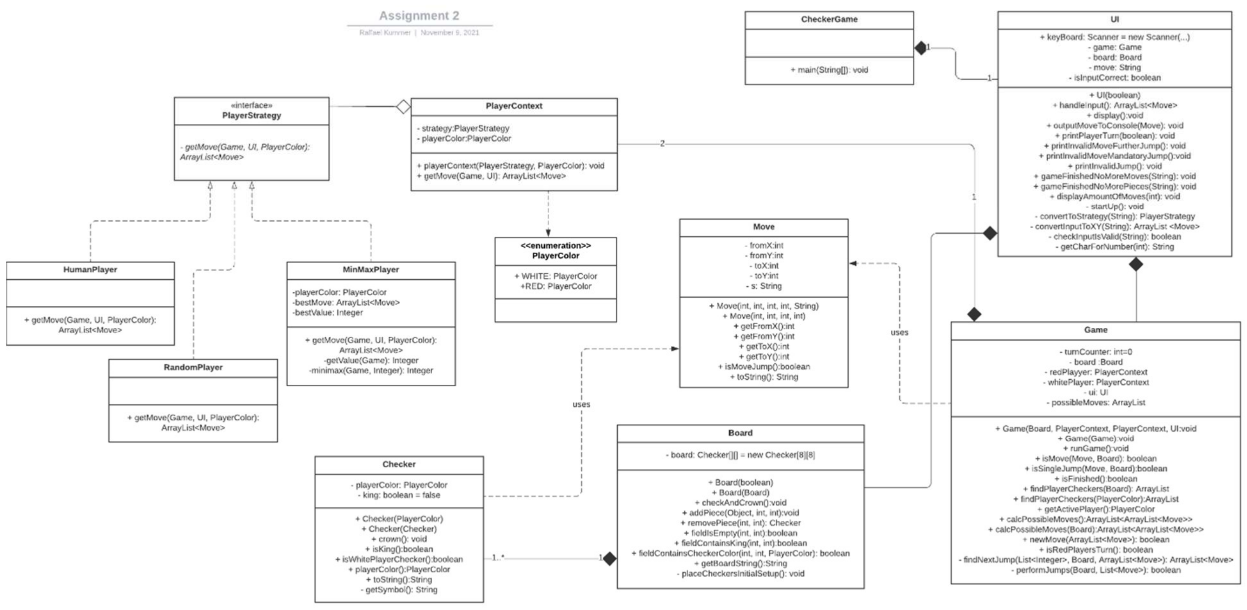
Assignment 2
Raffael Kummer | November 9, 2021
«interface»
PlayerStrategy
- getMove(Game, UI, PlayerColor): ArrayList<Move>
PlayerContext
- strategy:PlayerStrategy
- playerColor:PlayerColor
+ playerContext(PlayerStrategy, PlayerColor): void
+ getMove(Game, UI): ArrayList<Move>
2
1
CheckerGame
+ main(String[]): void
1
1
UI
+ keyBoard: Scanner = new Scanner(...)
- game: Game
- board: Board
- move: String
- isInputCorrect: boolean
+ UI(boolean)
+ handleInput(): ArrayList<Move>
+ display():void
+ outputMoveToConsole(Move): void
+ printPlayerTurn(boolean): void
+ printInvalidMoveFurtherJump(): void
+ printInvalidMoveMandatoryJump():void
+ printInvalidJump(): void
+ gameFinishedNoMoreMoves(String): void
+ gameFinishedNoMorePieces(String): void
+ displayAmountOfMoves(int): void
- startUp(): void
- convertToStrategy(String): PlayerStrategy
- convertInputToXY(String): ArrayList <Move>
- checkInputIsValid(String): boolean
- getCharForNumber(int): String
HumanPlayer
+ getMove(Game, UI, PlayerColor): ArrayList<Move>
RandomPlayer
+ getMove(Game, UI, PlayerColor): ArrayList<Move>
MinMaxPlayer
-playerColor: PlayerColor
-bestMove: ArrayList<Move>
-bestValue: Integer
+ getMove(Game, UI, PlayerColor): ArrayList<Move>
-getValue(Game): Integer
-minimax(Game, Integer): Integer
<<enumeration>>
PlayerColor
+ WHITE: PlayerColor
+RED: PlayerColor
Move
- fromX:int
- fromY:int
- toX:int
- toY:int
- s: String
+ Move(int, int, int, int, String)
+ Move(int, int, int, int)
+ getFromX():int
+ getFromY():int
+ getToX():int
+ getToY():int
+ isMoveJump():boolean
+ toString(): String
uses
uses
Game
- turnCounter: int=0
- board :Board
- redPlayyer: PlayerContext
- whitePlayer: PlayerContext
- ui: UI
- possibleMoves: ArrayList
+ Game(Board, PlayerContext, PlayerContext, UI:void
+ Game(Game):void
+ runGame():void
+ isMove(Move, Board): boolean
+ isSingleJump(Move, Board):boolean
+ isFinished():boolean
+ findPlayerCheckers(Board): ArrayList
+ findPlayerCheckers(PlayerColor):ArrayList
+ getActivePlayer():PlayerColor
+ calcPossibleMoves():ArrayList<ArrayList<Move>>
+ calcPossibleMoves(Board):ArrayList<ArrayList<Move>>
+ newMove(ArrayList<Move>): boolean
+ isRedPlayersTurn(): boolean
- findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
- performJumps(Board, List<Move>): boolean
Board
- board: Checker[][] = new Checker[8][8]
+ Board(boolean)
+ Board(Board)
+ checkAndCrown():void
+ addPiece(Object, int, int):void
+ removePiece(int, int): Checker
+ fieldIsEmpty(int, int):boolean
+ fieldContainsKing(int, int):boolean
+ fieldContainsCheckerColor(int, int, PlayerColor): boolean
+ getBoardString():String
- placeCheckersInitialSetup(): void
Checker
- playerColor: PlayerColor
- king: boolean = false
+ Checker(PlayerColor)
+ Checker(Checker)
+ crown(): void
+ isKing():boolean
+ isWhitePlayerChecker():boolean
+ playerColor():PlayerColor
+ toString():String
- getSymbol(): String
1..*
1

Figure 2 UML Diagram