# Assignment 3 – Group 16

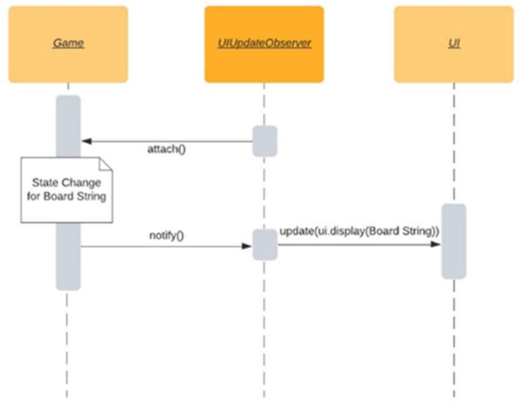
## 1. Two Design Patterns IN!

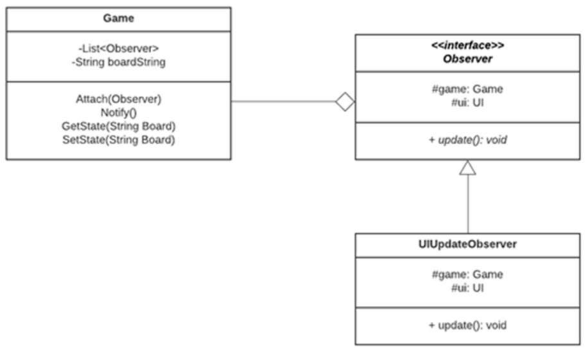
### 1.1 First pattern (Observer Pattern)

#### Why and how

Previously we had the modeling that the UI has a game, rather than being part of game. We now refactored so the logic is reversed and added the Observer Pattern. Here the UI is the subscriber and is informed every time the UI needs to be updated. The board information is then passed from the Game class to the UI class.

#### Sequence & Class diagram


Game
UIUpdateObserver
UI
attach()
State Change for Board String
notify()
update(ui.display(Board String))

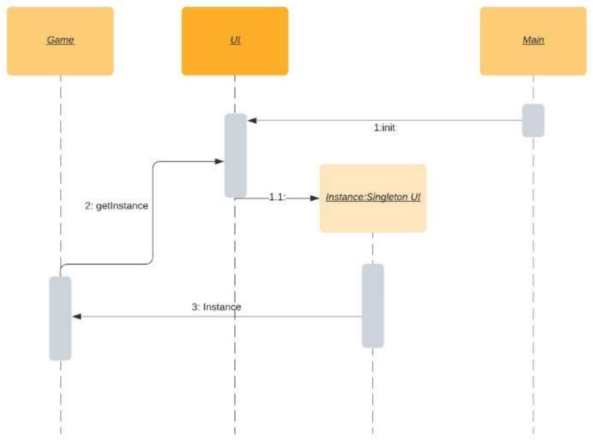


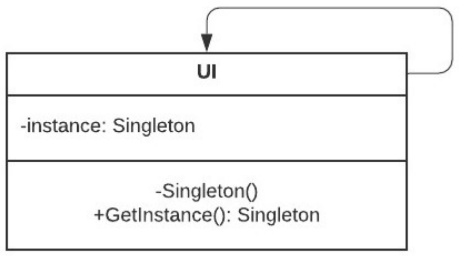
### 1.2 Second pattern (Singleton)

#### *Why and how*

Ensures only one instance of the UI class is generated. The UI instance is created in the main void in CheckerGame and whenever UI functionality is needed the getInstance() function returns the instance.

#### Sequence & Class diagram





## 2. Unit Tests (30 pts)

Consider ten important classes in your Checkers game.

The ten classes we chose are the following:

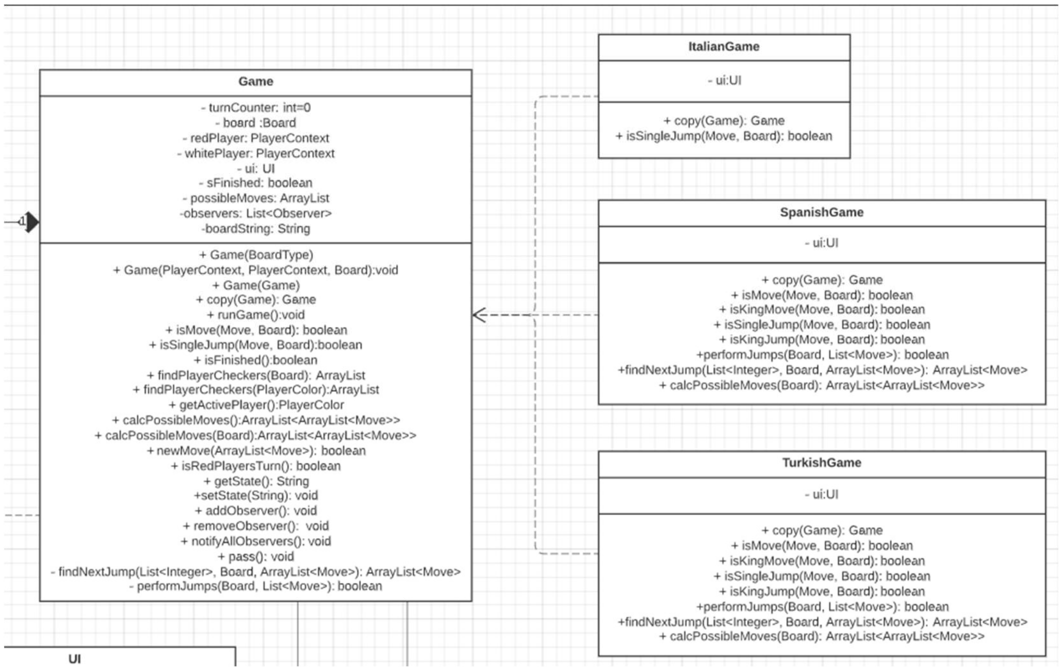
1. Game
2. ItalianGame
3. MinMaxPlayer
4. PlayerContext
5. RandomPlayer
6. SpanishGame
7. TurkishGame
8. Board
9. Checker
10. Move

We omitted the enums (PlayerColor and BoardType) from the analysis of the Line Coverage (and the testing) because they would be rather trivial and superfluous (since we would have to change the tests when we change the enums and this defeats the purpose of the unit tests).
We also did not include the HumanPlayer since we already have ten classes without it, and it is rather small. The UI does not appear either because we found it hard to test in a sensible way as there are a lot of print outs and interactions going on.

### 2.1 Importance of Classes current Responsibilities

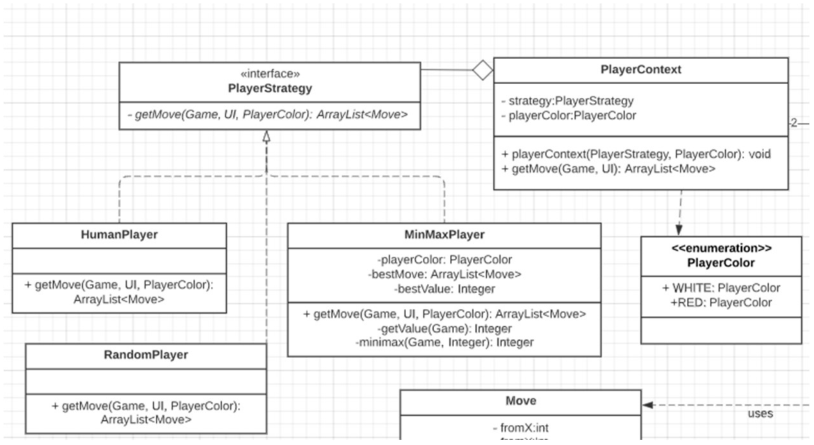
#### Game

The Game class implements the logic for a standard checkers game. SpanishGame, TurkishGame and ItalianGame extend the Game class and implement the specific rules for these types of games. These classes are crucial for our System and must work correctly.


Game
- turnCounter: int=0
- board :Board
- redPlayer: PlayerContext
- whitePlayer: PlayerContext
- ui: UI
- sFinished: boolean
- possibleMoves: ArrayList
-observers: List<Observer>
-boardString: String
+ Game(BoardType)
+ Game(PlayerContext, PlayerContext, Board):void
+ Game(Game)
+ copy(Game): Game
+ 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
+ getState(): String
+setState(String): void
+ addObserver(): void
+ removeObserver(): void
+ notifyAllObservers(): void
+ pass(): void
- findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
- performJumps(Board, List<Move>): boolean
1
UI
ItalianGame
- ui:UI
+ copy(Game): Game
+ isSingleJump(Move, Board): boolean
SpanishGame
- ui:UI
+ copy(Game): Game
+ isMove(Move, Board): boolean
+ isKingMove(Move, Board): boolean
+ isSingleJump(Move, Board): boolean
+ isKingJump(Move, Board): boolean
+performJumps(Board, List<Move>): boolean
+findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
+ calcPossibleMoves(Board): ArrayList<ArrayList<Move>>
TurkishGame
- ui:UI
+ copy(Game): Game
+ isMove(Move, Board): boolean
+ isKingMove(Move, Board): boolean
+ isSingleJump(Move, Board): boolean
+ isKingJump(Move, Board): boolean
+performJumps(Board, List<Move>): boolean
+findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
+ calcPossibleMoves(Board): ArrayList<ArrayList<Move>>

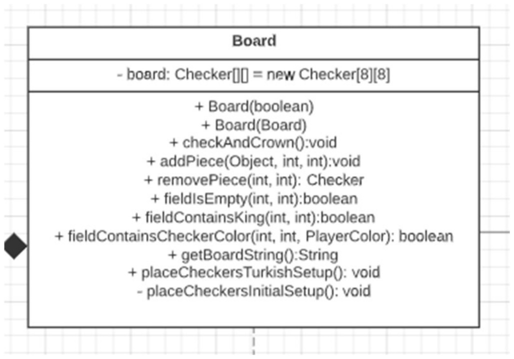
## PlayerContext

Using the strategy pattern, we implemented different types of players for our game. Focusing on the AI players (MinMax & RandomPlayer), these implementations need to be thoroughly tested to verify correct moves.


«interface»
PlayerStrategy
- getMove(Game, UI, PlayerColor): ArrayList<Move>
PlayerContext
- strategy:PlayerStrategy
- playerColor:PlayerColor
+ playerContext(PlayerStrategy, PlayerColor): void
+ getMove(Game, UI): ArrayList<Move>
2
HumanPlayer
+ 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
RandomPlayer
+ getMove(Game, UI, PlayerColor): ArrayList<Move>
Move
- fromX:int
uses

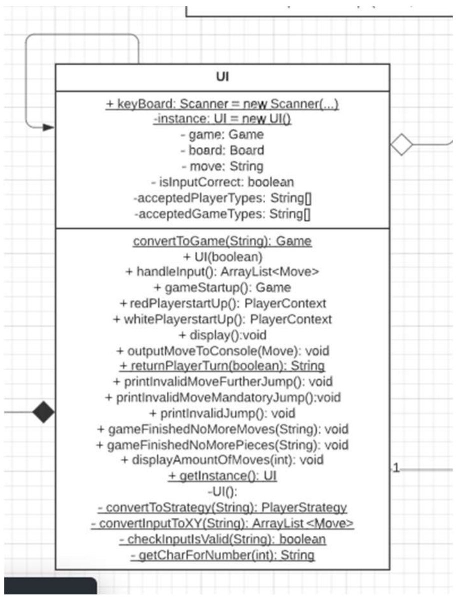
## Board

Related to the Game types, we need to verify if the Board setup is correct for the specific game type. Furthermore verifying if the board operations are working correctly.


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
+ placeCheckersTurkishSetup(): void
- placeCheckersInitialSetup(): void

## UI

Important for testing are the conversion functions in the UI class and checking if the inputs are valid inputs.


UI
+ keyBoard: Scanner = new Scanner(...)
-instance: UI = new UI()
- game: Game
- board: Board
- move: String
- isInputCorrect: boolean
-acceptedPlayerTypes: String[]
-acceptedGameTypes: String[]
convertToGame(String): Game
+ UI(boolean)
+ handleInput(): ArrayList<Move>
+ gameStartup(): Game
+ redPlayerstartUp(): PlayerContext
+ whitePlayerstartUp(): PlayerContext
+ display():void
+ outputMoveToConsole(Move): void
+ returnPlayerTurn(boolean): String
+ printInvalidMoveFurtherJump(): void
+ printInvalidMoveMandatoryJump():void
+ printInvalidJump(): void
+ gameFinishedNoMoreMoves(String): void
+ gameFinishedNoMorePieces(String): void
+ displayAmountOfMoves(int): void
+ getInstance(): UI
-UI():
- convertToStrategy(String): PlayerStrategy
- convertInputToXY(String): ArrayList <Move>
- checkInputIsValid(String): boolean
- getCharForNumber(int): String
1

For a complete overview of the responsibilities see section 3.a CRC cards.

## 2.2 Testing report

The line coverage overall is 88%. The respective values for our chosen classes can be seen on Figure 1.

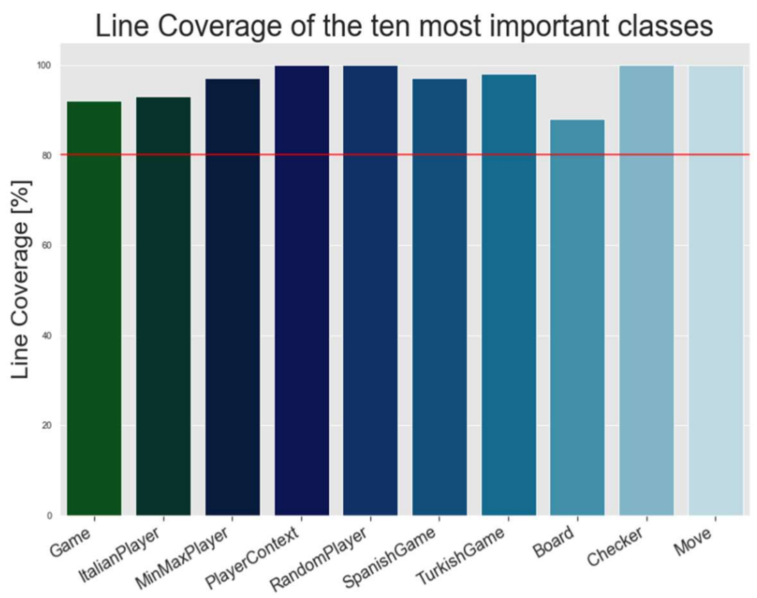
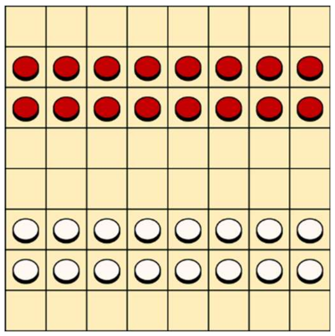


*Figure 1 Chart depicting the line coverage of the ten chosen classes*

## 3. 20%-Time (30 pts)

At the start of the game, the user can choose to play with standard rules, or one of the following checkers variations:

- Italian Checkers
  In addition to the standard rules, regular checkers pieces are not allowed to capture kings.
- Spanish Checkers
  Like Italian checkers and kings can make jumps to any square along a diagonal which contains only one opposing piece. The opposing piece doesn't need to be adjacent to the king, and the ending square of the move doesn't need to be adjacent to the captured piece. Kings can also move to any square along a free diagonal, if there are no jumps possible.
- Turkish Checkers
  This version is also played on a 8x8 board, but the number of checkers, starting position and moves change. The board is setup in in the following manner:



  Checkers move orthogonally instead of diagonally. Regular pieces can move forwards or sideways, but only kings can also move backwards. Otherwise, the rules are identical to Spanish Checkers: kings can move and jump to any free square along an orthogonal line, given that the line is empty or contains only one opponent piece, respectively.

## a. CRC cards

| Game | |
|---|---|
| Responsibility | Collaboration |
| - turnCounter<br>- check if finished<br>- Create and run Game<br>- notify observers of changes<br>- check legality of move<br>- check if single jump<br>- return active player<br>- find next jump<br>- calculate and perform possible moves | Board<br>PlayerContext<br>UI |

| HumanPlayer | |
|---|---|
| Responsibility | Collaboration |
| - Get input of human player via console | Move<br>UI<br>PlayerColor |

| ItalianGame | |
|---|---|
| Responsibility | Collaboration |
| - Create and return instance of an Italian game<br>- Checks if move is a single jump | Board<br>Move<br>PlayerColor<br>UI |

| MinMaxPlayer | |
|---|---|
| Responsibility | Collaboration |
| Simulates player with minmax-strategy | Move<br>PlayerColor<br>UI |

| PlayerContext | |
|---|---|
| Responsibility | Collaboration |
| Creates a PlayerContext with a strategy and color | Move<br>PlayerColor<br>UI |

| RandomPlayer | |
|---|---|
| Responsibility | Collaboration |
| Simulates a player with random (yet valid) moves | Move<br>PlayerColor<br>UI |

| SpanishGame | |
|---|---|
| Responsibility | Collaboration |
| Modifies the Game class such that it becomes the Spanish version of Checker | Board<br>Move<br>PlayerColor<br>UI<br>Checker |

| TurkishGame | |
|---|---|
| Responsibility | Collaboration |
| Modifies the Game class such that it becomes the Turkish version of Checker | Board<br>Move<br>PlayerColor<br>UI<br>BoardType<br>Checker |

| Board | |
|---|---|
| Responsibility | Collaboration |
| -Sets up the initial state of the board<br>-Add, remove and crown pieces<br>-check if field is empty, contains king or a certain color<br>-returns the whole board as a string | |

| Checker | |
|---|---|
| Responsibility | Collaboration |
| King<br>Crown<br>Color<br>Symbol (e.g. R_P) | |

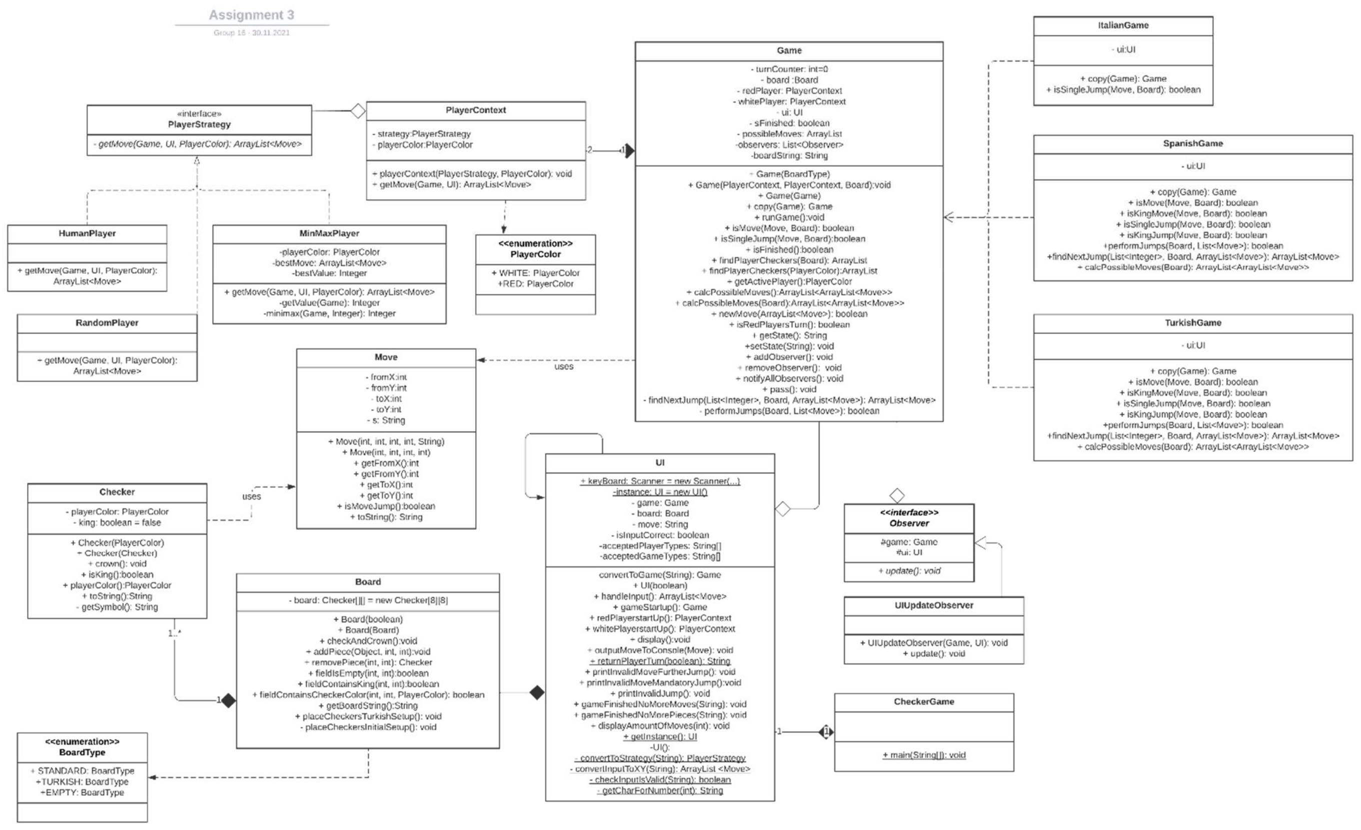
| Move | |
|---|---|
| Responsibility | Collaboration |
| Make a move<br>Print out move as string<br>Check if it is a move | |

| Observer | |
|---|---|
| Responsibility | Collaboration |
| Interface | Game<br>UI |

| UI | |
|---|---|
| Responsibility | Collaboration |
| Visualize the Checkers Game in console/terminal<br>Check input<br>Convert input into moves<br>Start up Game<br>Set up Players<br>Set strategy<br>Provide feedback if jump/move is invalid<br>Amount of moves<br>Game finished | Game<br>HumanPlayer<br>ItalianGame<br>MinMaxPlayer<br>PlayerContext<br>RandomPlayer<br>SpanishGame<br>TurkishGame |

| UIUpdateObserver | |
|---|---|
| Responsibility | Collaboration |
| Observes game for changes and updates UI accordingly | Game<br>UI |

## b. UML



Assignment 3
Group 16 - 30.11.2021
«interface»
PlayerStrategy
- getMove(Game, UI, PlayerColor): ArrayList<Move>
PlayerContext
- strategy:PlayerStrategy
- playerColor:PlayerColor
+ playerContext(PlayerStrategy, PlayerColor): void
+ getMove(Game, UI): ArrayList<Move>
2
1
HumanPlayer
+ 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
RandomPlayer
+ getMove(Game, UI, PlayerColor): ArrayList<Move>
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
Checker
- playerColor: PlayerColor
- king: boolean = false
+ Checker(PlayerColor)
+ Checker(Checker)
+ crown(): void
+ isKing():boolean
+ playerColor():PlayerColor
+ toString():String
- getSymbol(): String
uses
1..*
1
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
+ placeCheckersTurkishSetup(): void
- placeCheckersInitialSetup(): void
<<enumeration>>
BoardType
+ STANDARD: BoardType
+TURKISH: BoardType
+EMPTY: BoardType
Game
- turnCounter: int=0
- board :Board
- redPlayer: PlayerContext
- whitePlayer: PlayerContext
- ui: UI
- sFinished: boolean
- possibleMoves: ArrayList
-observers: List<Observer>
-boardString: String
+ Game(BoardType)
+ Game(PlayerContext, PlayerContext, Board):void
+ Game(Game)
+ copy(Game): Game
+ 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
+ getState(): String
+setState(String): void
+ addObserver(): void
+ removeObserver(): void
+ notifyAllObservers(): void
+ pass(): void
- findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
- performJumps(Board, List<Move>): boolean
ItalianGame
- ui:UI
+ copy(Game): Game
+ isSingleJump(Move, Board): boolean
SpanishGame
- ui:UI
+ copy(Game): Game
+ isMove(Move, Board): boolean
+ isKingMove(Move, Board): boolean
+ isSingleJump(Move, Board): boolean
+ isKingJump(Move, Board): boolean
+performJumps(Board, List<Move>): boolean
+findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
+ calcPossibleMoves(Board): ArrayList<ArrayList<Move>>
TurkishGame
- ui:UI
+ copy(Game): Game
+ isMove(Move, Board): boolean
+ isKingMove(Move, Board): boolean
+ isSingleJump(Move, Board): boolean
+ isKingJump(Move, Board): boolean
+performJumps(Board, List<Move>): boolean
+findNextJump(List<Integer>, Board, ArrayList<Move>): ArrayList<Move>
+ calcPossibleMoves(Board): ArrayList<ArrayList<Move>>
UI
+ keyBoard: Scanner = new Scanner(...)
-instance: UI = new UI()
- game: Game
- board: Board
- move: String
- isInputCorrect: boolean
-acceptedPlayerTypes: String[]
-acceptedGameTypes: String[]
convertToGame(String): Game
+ UI(boolean)
+ handleInput(): ArrayList<Move>
+ gameStartup(): Game
+ redPlayerstartUp(): PlayerContext
+ whitePlayerstartUp(): PlayerContext
+ display():void
+ outputMoveToConsole(Move): void
+ returnPlayerTurn(boolean): String
+ printInvalidMoveFurtherJump(): void
+ printInvalidMoveMandatoryJump():void
+ printInvalidJump(): void
+ gameFinishedNoMoreMoves(String): void
+ gameFinishedNoMorePieces(String): void
+ displayAmountOfMoves(int): void
+ getInstance(): UI
-UI():
- convertToStrategy(String): PlayerStrategy
- convertInputToXY(String): ArrayList <Move>
- checkInputIsValid(String): boolean
- getCharForNumber(int): String
<<interface>>
Observer
#game: Game
#ui: UI
+ update(): void
UIUpdateObserver
+ UIUpdateObserver(Game, UI): void
+ update(): void
1
1
CheckerGame
+ main(String[]): void

*Figure 2 UML Diagram (Link: https://lucid.app/lucidchart/45996cc8-72dc-4548-8da1-fb9d9bc59a4c/edit?viewport_loc=-149%2C-390%2C2261%2C1845%2CHWEp-vi-RSFO&invitationId=inv_b0eab0bf-dd53-4d58-a670-19ec47a886e2)*