Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959

Software Construction HS 2021 Group 03: Assignment 3

General notes:

At the very end of the assignment an overview of the current implementation with all design patterns and new game mechanics is shown in an UML class diagram (Figure 11). During this report excerpts may show parts of said diagram.
As one can notice in our Github repository we decided to divide Assignment 3 in two separate folders: “sc_assignment_3” shows the code relevant for Exercise 2 since testing was based on the implementation before Exercise 1 and 3 were tackled (as a side note, the Strategy Pattern is also implemented in the “sc_assignment_3” folder). On the other hand “sc_assignment_3_2” shows the code for the finished Assignment without the test package.

Exercise 1:

For this first exercise, the two chosen Design Patterns were once the Singleton Design Pattern, and secondly, the Strategy Design Pattern.

Strategy Design Pattern:

(1) In our existing implementation, we strongly depend on the game and the Validator classes. The Game class has to ask the Validator to check for the specific moves to be valid or to do a particular calculation (e.g., calcMoveSpace). Those callings of the Validator methods show that the Game logic is mainly implemented in the Validator class. Therefore the Game needs the existence of the Validator class for every move to know if it is valid. However, this makes it more challenging to extend the Game logic or even create a different game logic on the Checkers game. Therefore, to increase the polymorphism of our code, we decided to implement a Strategy Pattern for the Validator class. At the moment we only have created a framework with the Strategy Pattern. Hence this framework can be used to extend the code. This is as we only have one validation class. However in further extension one can use this pattern and add another validation class. Thereby through the Strategy Pattern it is possible to change the Game logic at run time. For example one could change between the normal checker Game logic and our extension containing the queen.

This implementation was done by using an interface, such that two different Validators have the same contract and are of the same type, “IValidator”. Therefore this allows loose coupling as the game does not have to know which Validator is used and how the method is implemented. Additionally, we split the interfaces into three distinct interfaces. First, we constructed the “ICheckValidator” interface. This interface includes all the methods that do checks and return a boolean type. The second interface implemented was the “ICalcValidator”; this interface contains all the calculations that have to be done in the checkers Game like for example the calculations to find out which pieces can move or jump. Finally, the third interface “IValdiator” extends the “ICalcValidator” and the “ICheckValidator” interface. This separation was done to have a clean-cut between the responsibilities that the Validator has to do. In the end, the Validator implements the “IValidator” interface. Additional changes that had to be made correspond to the fact that the Validator object is now of the type “IValidator”.

(2) Firstly, the sequence that is used is a normal sequence that occurs if any arbitrary piece is moved. Thereby, the following sequence diagram mainly shows the interaction between the User, the Game, and the Validator class. The additional classes are classes to give a full overview of the sequence. In this sequence diagram, only one concrete Validator will be shown (whereby through the blue line a possible interaction will be shown). This is as only one concrete Validator was implemented. Secondly, we assume that either one Validator or another Validator will be active at any specific time. This is as one either wants one specific game logic or the other. Nevertheless the strategy pattern allows one to change between different Validators at run time. Therefore it is possible to, for example, start off with a normal checkers Game logic, and then switch at run time to our special checker Game logic which includes the queen.

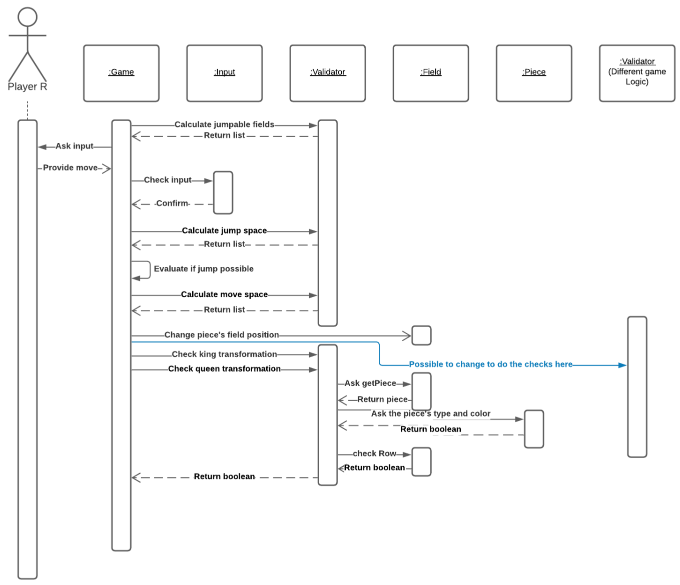

Player R
:Game
:Input
:Validator
:Field
:Piece
:Validator (Different game Logic)
Calculate jumpable fields
Return list
Ask input
Provide move
Check input
Confirm
Calculate jump space
Return list
Evaluate if jump possible
Calculate move space
Return list
Change piece's field position
Possible to change to do the checks here
Check king transformation
Check queen transformation
Ask getPiece
Return piece
Ask the piece's type and color
Return boolean
check Row
Return boolean
Return boolean

Figure 1: UML Sequence Diagram of the Game Validator Communication (blue line shows how a potential second validator would interact)

(3) This UML class diagram, omits the methods and attributes of the classes Validator and Game, as it focuses on the implementation of the Strategy Design Pattern.

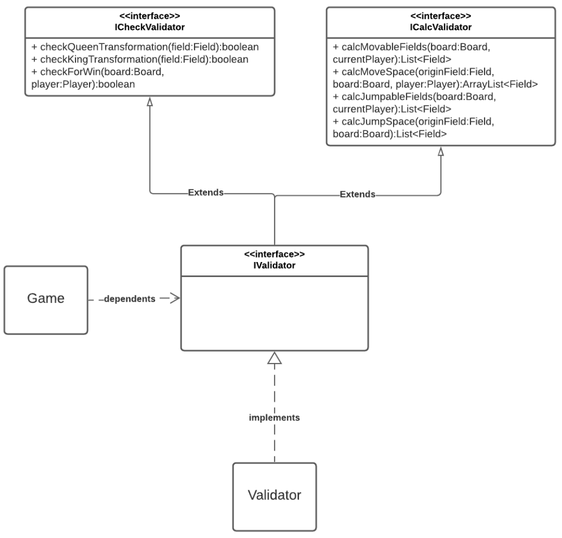

<<interface>>
ICheckValidator
+ checkQueenTransformation(field:Field):boolean
+ checkKingTransformation(field:Field):boolean
+ checkForWin(board:Board, player:Player):boolean
<<interface>>
ICalcValidator
+ calcMovableFields(board:Board, currentPlayer):List<Field>
+ calcMoveSpace(originField:Field, board:Board, player:Player):ArrayList<Field>
+ calcJumpableFields(board:Board, currentPlayer):List<Field>
+ calcJumpSpace(originField:Field, board:Board):List<Field>
Extends
Extends
<<interface>>
IValidator
Game
dependents
implements
Validator

Figure 2: UML Class Diagram of the Game Validator Communication

Singleton Design Pattern:

(1) The Board is heavily used: it is accessed by the Launcher, the Game, the Printer, and the Validator. To ensure at all times that the same instance is used by the different parts of the code that are interested in the Board, the Singleton Design Pattern was applied to it. It guarantees that there is only a single instance of the Board.
To achieve that the Board's constructor was made private. That way it is only accessible inside the Board's class. To get the one and only instance of the Board from the outside a public static method getInstance(int rows, int columns) was introduced, that checks whether an instance of the Board already exists, if not it will create one, and returns the instance afterwards. If the instance already exists, it will return this specific instance. If another developer decides to work on this checker game, it will be seen easily that the Board is meant to be a Singleton and that it is not possible to create a second instance.

(2) Sequence diagram of how the Singleton works dynamically in our code.

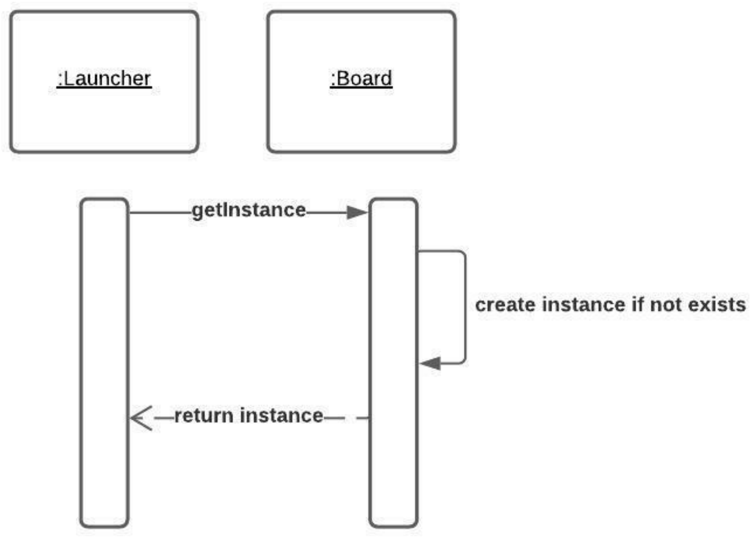


Figure 3: UML Sequence Diagram of the Singleton Design Pattern

(3) Figure 4 shows an excerpt from the "huge" class diagram that shows the relationships between all our classes, Figure 11. The excerpt shows in detail that the Board class contains a private static instance variable INSTANCE and a public method getInstance(int rows, int columns) now.

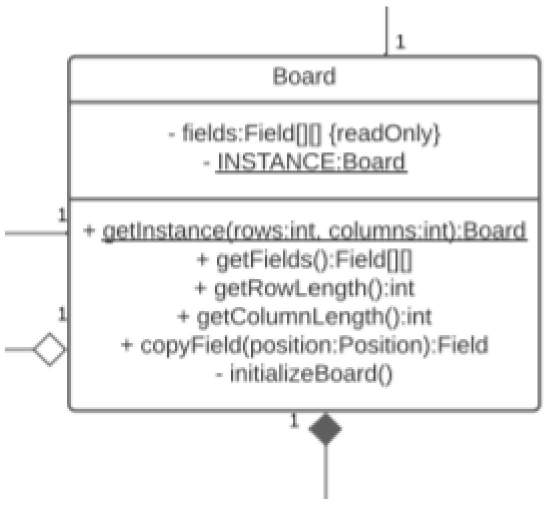


Figure 4: UML Class Diagram of the Board Class, excerpt from Figure 11

During the startup the Launcher calls the getInstance(int rows, int columns) method, the Board will check whether an instance already exists and return an instance of itself. For now, the getInstance(int rows, int columns) method is not used a second time, but the design of the Board was made more robust for future possible use cases.

Exercise 2:

2.1

For our checkers Game, we have precisely ten classes (before completing exercise 1 and 3). Thereby, we considered nine of these classes to be of such importance to write unit tests. During this procedure, those nine classes were grouped into three subgroups according to their responsibilities. The three responsibilities are:

1. Checks for the game logic & and game moves
2. Checks for the user interaction
3. Checks for classes that hold information and needs to give it back

These three subgroups are of particular importance, as they contain all the functionality and the data that a checkers game needs.

The first subgroup contains the classes of the Game and the Validator. The UML class diagram (Figure 5) shows that these classes are the heart of the checkers game. As one can see in the CRC Cards, the responsibility of the Game class is to keep track of the Player's turn, know the move and jump spaces, and be able to execute a move. This shows that the Game class is at the center of the game execution.
Additionally, the main responsibility of the Validator class is to maintain the game logic and let the game operate on this logic. Thereby the Validator checks for the Type of the Piece, checks and calculates the move- and jump-spaces. Additionally, as the Validator knows the move and jump space, it can check for a winner. This shows that the Validator is an essential class in our system design, as the game logic must be correct to have a well-behaved game.

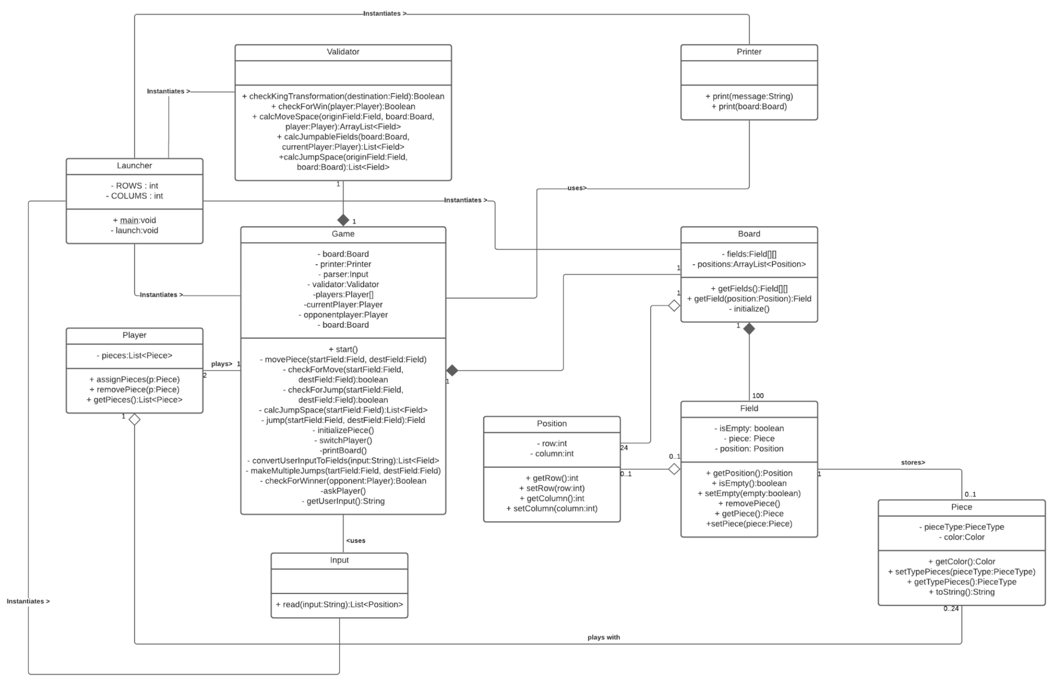


Figure 5: UML Class Diagram

The second subgroup, checking for user interaction, contains the "IO" classes, namely Input and Printer. Thereby the responsibilities of the Input class are, as it suggests, to get the users input from the system's terminal and then restructure this input in such a way that the rest of the program can make sense of it. This also shows the importance of the class. The Input class has to know if an input is valid or not and act respectively. Therefore it is vital to a functioning logic and is the foundation of any move(-check) in the game. As one can see in Figure 5 the Input class works directly with the Game class which even further promotes its importance in the game.
The second class in this subgroup is the Printer class. The Printer is responsible for printing an updated version of the Board each time a Piece was moved, to the console. Therefore this class is important, as it has to deliver a correct visual representation of a particular instance of the Board. Same as Input the Printer receives instructions from the game and thereby is directly connected to the heart of the checkers game.

The third subgroup are classes that hold and maintain the data so that other classes can make use of it. Consequently, for these classes, one has to check if the classes' data can be messaged in a valid way to the other classes. The classes that fall under this subgroup are Field, Board, Piece, Player, Position. Those five classes mainly store information that other classes need to use.
Firstly the responsibility of the Field class is to know and maintain if a Piece is on that particular field on the Board. Thereby, it also has the responsibility to add or remove a piece at that specific instant of field. It also has to know which row it is to maintain with the check if it is in the first or eighth row.
Another class is the Board class. Its responsibility is to initialize the Board and maintain the Board (the board is a 2D array of Fields). This is done in such a way that it always knows the position of the pieces on the Board.

A third class that has the primary responsibility to maintain the data is the Piece class. The Piece class has the responsibility to know its color and its Type. Additionally, it is also responsible for maintaining the Type that it has at a specific instance in time.
A further class that falls under this category is the Player class. The Player class is responsible for keeping track and maintaining the pieces that belong to a specific player on the Board. Thereby, it needs to know different attributes of the Pieces that it owns. For example, the Player needs to know what color of pieces corresponds to himself, the number of pieces he still owns.
The last class that corresponds to this subgroup is the Position class. As the name suggests, this class maintains the position of a Field/Piece on the Board. Thereby its main purpose is to know the row and column it is. It also maintains the knowledge if it is positioned in the first row (edgeRed) or the last row (edgeWhite).

As these are only nine classes we test for, we left out the Launcher class. This class was considered less important from a testing point of view, as this class only initiates the checkers game. Thereby it has the responsibility to instantiate all the crucial classes to get the game started in a reasonable manner. Consequently, it has no data or underlying logic to test for. Additionally, while "black testing" and running the code, the tester should be able to notice if there is a bug in the instantiation (e.g., the Board is 10x10 and not 8x8 in the terminal).

2.2

Some classes were easy to test because their methods cover a very simple implementation, for example the "pawnToKing" method which changes the PieceType.

The Printer class was a first hurdle because initially we tried to test the console output of the 2D-representation of the Board. This proved to be rather difficult and tedious which is why we simplified the class by changing the building of the Board to return to a new method which then prints it to the console. That way we have a StringBuilder returned to the newly created method and testing was simplified immensely. We merely tested the "skeleton" of the Printer function since implementation and correct positioning of Pieces was tested in GameTest and ValidatorTest.

Unfortunately we noticed that the Game class implementation is lacking in some aspects. All variables and methods with the exception of the start() method were set to private. That made testing pretty much impossible. As a workaround we decided to make some methods public and also initialize the "currentPlayer" and "opponentPlayer" in the Game instance rather than the "start()" method. That way they are instantiated with a new Game instance and there is no need to actually start the Game. Also the "makeMultipleJumps()" method is called after askPlayer() which means user input is required twice - at first in order to make a jump (if it is viable) and then to jump again. Simulating different user input for two different methods which call each other and need to follow the game logic was not implemented. This is an issue which should have been addressed during implementation in hindsight. However the Game class was still tested properly and with a few adjustments most methods were covered. The takeaway for us was that making everything within a class private and inaccessible is not always a great idea especially for testing.

2.3

During our testing, we not only wanted to achieve the highest possible line coverage. In addition to a high line coverage, we also tried to test the different classes in multiple scenarios, to also cover corner cases, where the program could misbehave. During our testing, we achieved a total line coverage of 91,61%. In Table 1 the distribution of the line coverage can be seen. Thereby one can notice that all line coverages are above 80%, whereby the 6 out of the 9 classes under test have a line coverage of 100%. Additionally, we also created a histogram (Figure 6) to visualize this table.

Classes	Line Coverage	Line Covered	Total Lines
Game	80,67%	96	119
Validator	100,00%	106	106
Input	100,00%	15	15
Printer	86,36%	19	22
Field	100,00%	13	13
Board	100,00%	13	13
Piece	92,31%	12	13
Player	100,00%	9	9
Position	100,00%	12	12
Total	91,61%	295	322

Table 1: Displays the numbers of the line coverage, including the total line coverage.

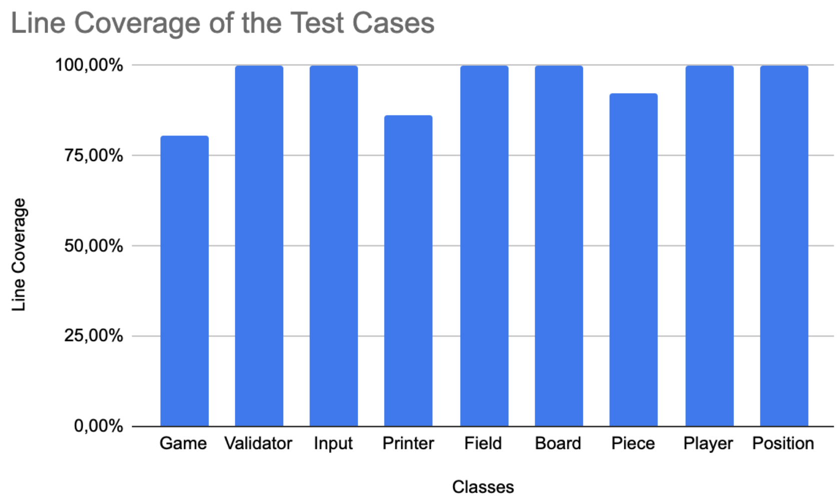


Figure 6: Histogram showing the line coverage of the classes

Exercise 3:

3.1

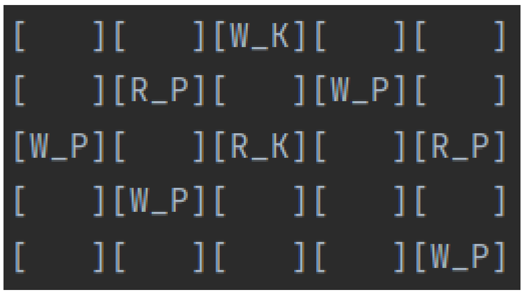
We include two additional functions: “help” and “auto”. They work as follows:

1. If the player types “help” as an input, all the pieces that can be manipulated based on rules in the current turn will be printed in the console. When the jump is possible, the output will be the piece(s) that can take a jump, otherwise, the piece(s) that can take a normal move will be given. In multiple jumps, only this concerned piece will be shown.
2. If the player types “auto” as an input, a jump or a move will be chosen automatically, the board after being applied this operation will be printed in the console immediately.
3. The actions chosen by “auto” are rule-based. When the jump is possible, only a jump action can happen, otherwise a normal move, then switch to the opponent’s turn if multiple jumps are not possible. If the player is taking multiple jumps, the “auto” also can be used. In this situation, the “auto” will continue to manipulate this concerned piece and won’t end the current player’s turn until the end. The “auto” only takes one action at a time and can be repeatedly used in multiple jumps.
4. The “auto” is heuristic-based which means the actions are not chosen randomly. The details are as follows:

   We calculate the heuristic scores for the current player after taking every possible action respectively and choose the one leading the highest score. This score is made up of two parts: s_1 encourages the actions of directly taking an opponent’s piece according to the priority of the piece type. We think a queen is worth more than a king and a king is worth more than a pawn. Let N be the number of current player’s pieces, n be the number of opponent’s pieces. P, K, and Q stand for pawn, king, and queen respectively, then

   $$s_1 = \left(N_P - n_P\right) \times 5 + \left(N_K - n_K\right) \times 10 + \left(N_Q - n_Q\right) \times 20$$

   The higher this score, the more pieces with more value we have compared to our opponent, the more likely we will win as a consequence. s_2 encourages the actions which make the concerned piece able to jump over a piece of opponent and prevent the jump of the opponent’s pieces in following turns. Let A be the set of current player’s pieces, B be the set of opponent’s pieces. J stands for the number of grids a piece can jump to, for example, J of R_K is 2 in the situation shown in Figure 7.



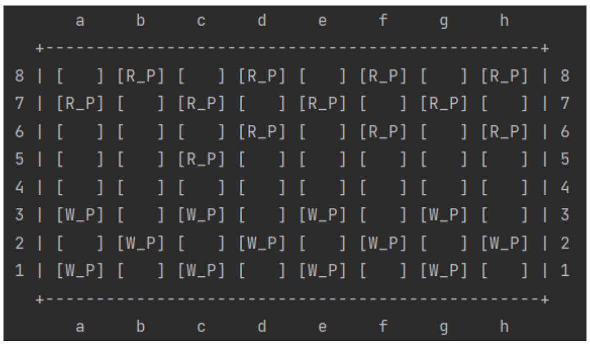
```
[   ][   ][W_K][   ][   ]
[   ][R_P][   ][W_P][   ]
[W_P][   ][R_K][   ][R_P]
[   ][W_P][   ][   ][   ]
[   ][   ][   ][   ][W_P]
```

Figure 7: Suited situation of an ongoing game

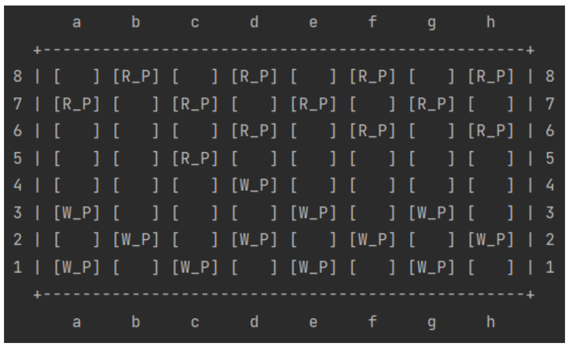
The higher s_2, the better positions our pieces occupy than our opponent, the more strategies and options we have to take the opponent's pieces in following turns. Finally, $s = s_1 + s_2$, where

$$s_2 = \sum_{i \in A} 4J_i - \sum_{i \in B} 5J_i$$

The following examples can illustrate the effect of our heuristic: As shown in Figure 8, when the white player uses "auto", the white piece at [c3] moves to [d4], where it can take the red piece at [c5] if this red piece stays the same position in the next turn. The white piece at [e3] can also move to [d4], but this action will make the red piece at [c5] able to take this white piece in the next turn.



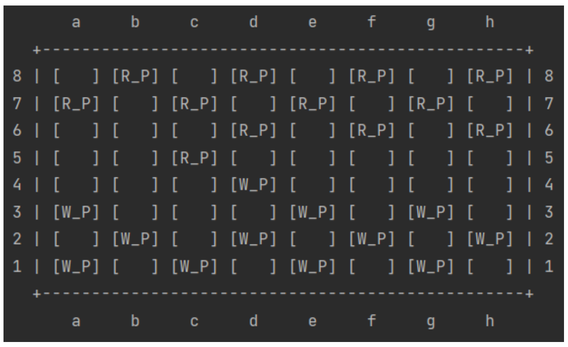
```
      a     b     c     d     e     f     g     h
  +-------------------------------------------------+
8 | [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 8
7 | [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] | 7
6 | [   ] [   ] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 6
5 | [   ] [   ] [R_P] [   ] [   ] [   ] [   ] [   ] | 5
4 | [   ] [   ] [   ] [   ] [   ] [   ] [   ] [   ] | 4
3 | [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] | 3
2 | [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] | 2
1 | [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] | 1
  +-------------------------------------------------+
      a     b     c     d     e     f     g     h
```



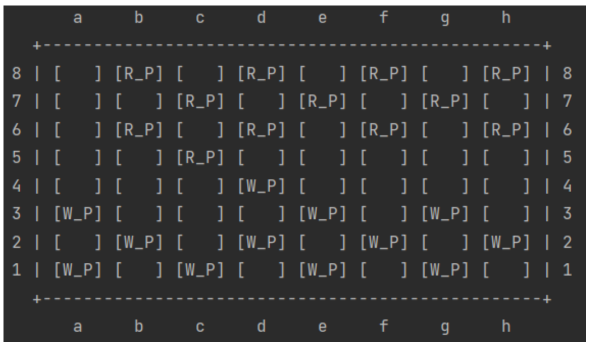
```
      a     b     c     d     e     f     g     h
  +-------------------------------------------------+
8 | [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 8
7 | [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] | 7
6 | [   ] [   ] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 6
5 | [   ] [   ] [R_P] [   ] [   ] [   ] [   ] [   ] | 5
4 | [   ] [   ] [   ] [W_P] [   ] [   ] [   ] [   ] | 4
3 | [W_P] [   ] [   ] [   ] [W_P] [   ] [W_P] [   ] | 3
2 | [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] | 2
1 | [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] | 1
  +-------------------------------------------------+
      a     b     c     d     e     f     g     h
```

Figure 8: Before and after the white player inputs "auto"

Then in the next turn of the red player, if "auto" is also applied, the result is shown in Figure 9. Moving the red piece at [c5] to [b4] or the red piece at [a7] to [b6] can avoid the red piece at [c5] being taken by the white piece at [d4] in the following white player's turn, but the action [c5]x[b4] makes this piece under the risk of being eaten by the white piece at [a3]. The "auto" chose the better action here.



```
      a     b     c     d     e     f     g     h
  +-------------------------------------------------+
8 | [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 8
7 | [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] | 7
6 | [   ] [   ] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 6
5 | [   ] [   ] [R_P] [   ] [   ] [   ] [   ] [   ] | 5
4 | [   ] [   ] [   ] [W_P] [   ] [   ] [   ] [   ] | 4
3 | [W_P] [   ] [   ] [   ] [W_P] [   ] [W_P] [   ] | 3
2 | [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] | 2
1 | [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] | 1
  +-------------------------------------------------+
      a     b     c     d     e     f     g     h
```



```
      a     b     c     d     e     f     g     h
  +-------------------------------------------------+
8 | [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 8
7 | [   ] [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] | 7
6 | [   ] [R_P] [   ] [R_P] [   ] [R_P] [   ] [R_P] | 6
5 | [   ] [   ] [R_P] [   ] [   ] [   ] [   ] [   ] | 5
4 | [   ] [   ] [   ] [W_P] [   ] [   ] [   ] [   ] | 4
3 | [W_P] [   ] [   ] [   ] [W_P] [   ] [W_P] [   ] | 3
2 | [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] | 2
1 | [W_P] [   ] [W_P] [   ] [W_P] [   ] [W_P] [   ] | 1
  +-------------------------------------------------+
      a     b     c     d     e     f     g     h
```

Figure 9: Before and after the red player inputs "auto"

These two "auto" actions are compatible with those obtained by employing the Minimax algorithm. Compared to the random choices from the possible actions, our heuristic is smart enough to select the better actions to help the player win.

<u>**3.2**</u>

From the perspective of responsibility driven design, we don’t need to change the design logic to implement these two features. Nonetheless out of clarity purposes we will include a revised version of outer CRC cards (Figure 10). The UML class diagram is shown in Figure 11, the newly added methods are marked as green, and the modified parts are marked as blue. Also the Launcher class was left out since its responsibility remains the instantiation of our main classes and does not contribute majorly to the understanding of our implementation.

For the “help” feature, we create the *printHelp()* method, and revise the *askPlayer()* to make it possible to react to the input of “help” instead of an action of move or jump. For the “auto” feature, our strategy is first obtaining the list of possible actions. Then for each alternative, we take this action first and compute the heuristic score for the current player, then we undo this particular action. Finally we choose the action leading the highest score and print out the board after applying this best action. Since when a jump is possible, the player cannot conduct a simple move, also the manipulations of a jump and a move are different, we create two methods *calcHeuristicJumpVal()* and *calcHeuristicMoveVal()* for these two different conditions. The method *makeAutoAction()* executes the chosen action automatically. We also revise the *makeMultipleJumps()* to implement the “auto” function in the series of multiple jumps.

To better undo the action, we create the methods *removePieceInBetween()* and *addPieceInBetween()*. The original methods *movePiece()* in Game class, *removePiece()* and *getPiece()* in Piece class are also modified accordingly. Finally, we add the methods to compute the number of pieces of different types for each player in Player class in order to facilitate the computation of heuristic, they are *getNumberOfPawns()*, *getNumberOfKings()*, and *getNumberOfQueens()*.

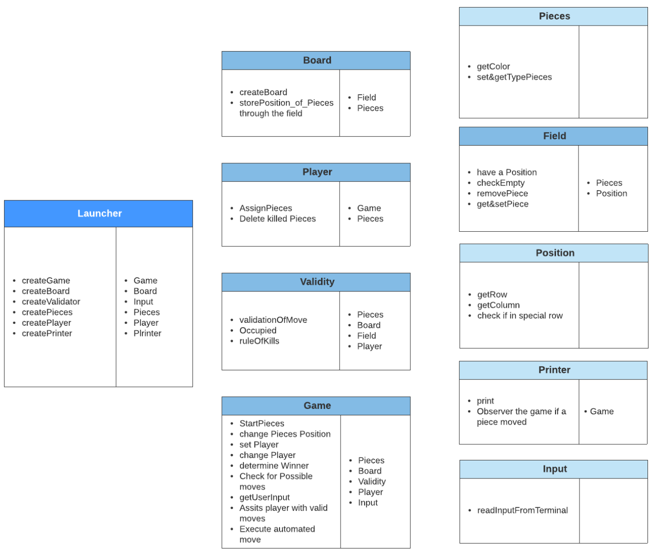
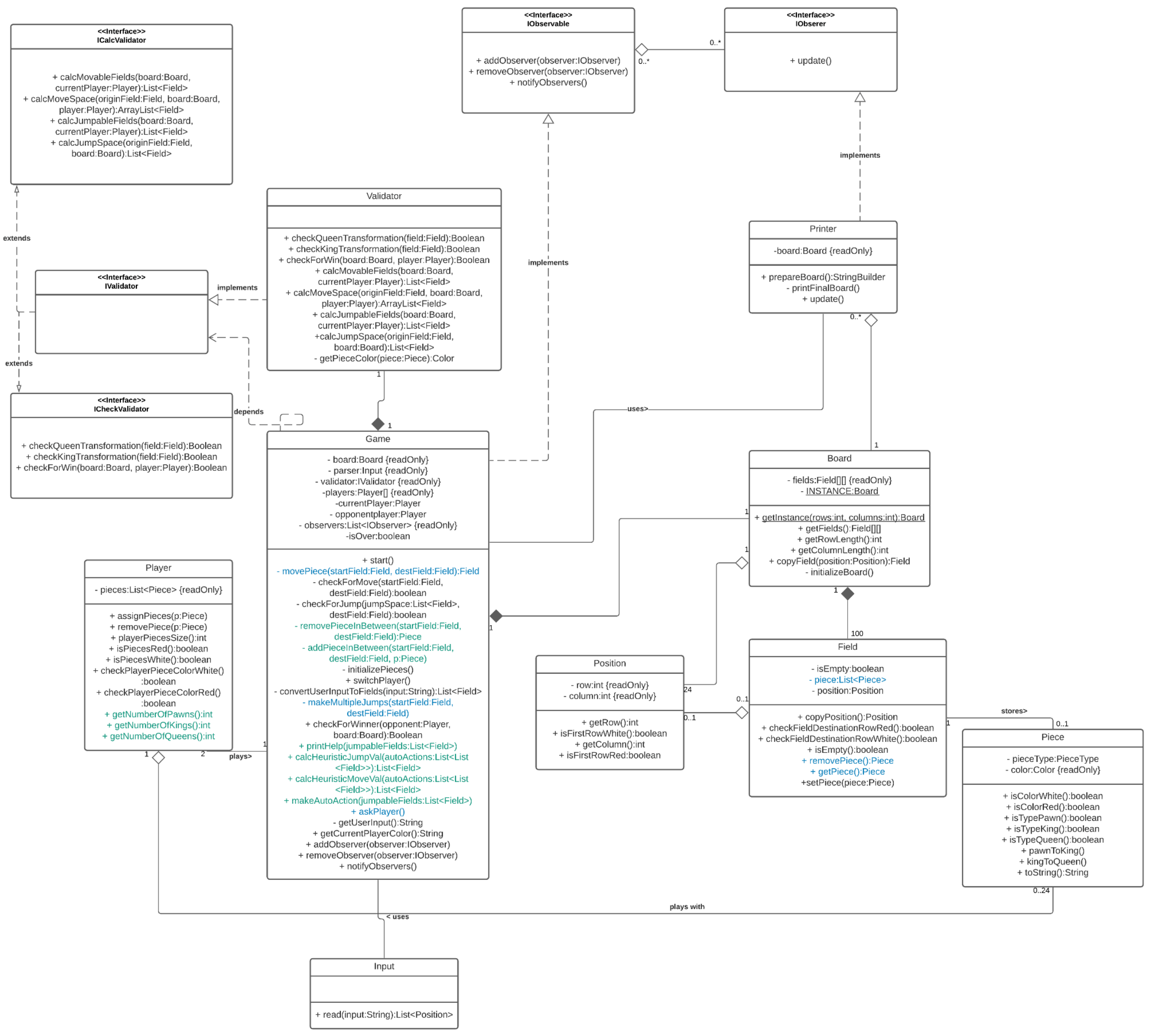


Figure 10: The Revised CRC cards

Kai Zinnhardt 19-763-176/ Jia Fu 21-907-662/ Paul Luley 21-741-491/ Joel Meier 16-916-959



<<Interface>>
ICalcValidator
+ calcMovableFields(board:Board, currentPlayer:Player):List<Field>
+ calcMoveSpace(originField:Field, board:Board, player:Player):ArrayList<Field>
+ calcJumpableFields(board:Board, currentPlayer:Player):List<Field>
+ calcJumpSpace(originField:Field, board:Board):List<Field>
<<Interface>>
IObservable
+ addObserver(observer:IObserver)
+ removeObserver(observer:IObserver)
+ notifyObservers()
0..*
0..*
<<Interface>>
IObserer
+ update()
implements
Validator
+ checkQueenTransformation(field:Field):Boolean
+ checkKingTransformation(field:Field):Boolean
+ checkForWin(board:Board, player:Player):Boolean
+ calcMovableFields(board:Board, currentPlayer:Player):List<Field>
+ calcMoveSpace(originField:Field, board:Board, player:Player):ArrayList<Field>
+ calcJumpableFields(board:Board, currentPlayer:Player):List<Field>
+calcJumpSpace(originField:Field, board:Board):List<Field>
- getPieceColor(piece:Piece):Color
extends
implements
implements
<<Interface>>
IValidator
extends
Printer
-board:Board {readOnly}
+ prepareBoard():StringBuilder
- printFinalBoard()
+ update()
0..*
<<Interface>>
ICheckValidator
+ checkQueenTransformation(field:Field):Boolean
+ checkKingTransformation(field:Field):Boolean
+ checkForWin(board:Board, player:Player):Boolean
depends
1
1
uses>
Game
- board:Board {readOnly}
- parser:Input {readOnly}
- validator:IValidator {readOnly}
-players:Player[] {readOnly}
-currentPlayer:Player
- opponentplayer:Player
- observers:List<IObserver> {readOnly}
-isOver:boolean
+ start()
- movePiece(startField:Field, destField:Field):Field
- checkForMove(startField:Field, destField:Field):boolean
- checkForJump(jumpSpace:List<Field>, destField:Field):boolean
- removePieceInBetween(startField:Field, destField:Field):Piece
- addPieceInBetween(startField:Field, destField:Field, p:Piece)
- initializePieces()
+ switchPlayer()
- convertUserInputToFields(input:String):List<Field>
- makeMultipleJumps(startField:Field, destField:Field)
+ checkForWinner(opponent:Player, board:Board):Boolean
+ printHelp(jumpableFields:List<Field>)
+ calcHeuristicJumpVal(autoActions:List<List<Field>>):List<Field>
+ calcHeuristicMoveVal(autoActions:List<List<Field>>):List<Field>
+ makeAutoAction(jumpableFields:List<Field>)
+ askPlayer()
- getUserInput():String
+ getCurrentPlayerColor():String
+ addObserver(observer:IObserver)
+ removeObserver(observer:IObserver)
+ notifyObservers()
1
Board
- fields:Field[][] {readOnly}
- INSTANCE:Board
+ getInstance(rows:int, columns:int):Board
+ getFields():Field[][]
+ getRowLength():int
+ getColumnLength():int
+ copyField(position:Position):Field
- initializeBoard()
1
1
1
Player
- pieces:List<Piece> {readOnly}
+ assignPieces(p:Piece)
+ removePiece(p:Piece)
+ playerPiecesSize():int
+ isPiecesRed():boolean
+ isPiecesWhite():boolean
+ checkPlayerPieceColorWhite():boolean
+ checkPlayerPieceColorRed():boolean
+ getNumberOfPawns():int
+ getNumberOfKings():int
+ getNumberOfQueens():int
1
2
plays>
1
1
100
Position
- row:int {readOnly}
- column:int {readOnly}
+ getRow():int
+ isFirstRowWhite():boolean
+ getColumn():int
+ isFirstRowRed:boolean
24
0..1
0..1
Field
- isEmpty:boolean
- piece:List<Piece>
- position:Position
+ copyPosition():Position
+ checkFieldDestinationRowRed():boolean
+ checkFieldDestinationRowWhite():boolean
+ isEmpty():boolean
+ removePiece():Piece
+ getPiece():Piece
+setPiece(piece:Piece)
stores>
1
0..1
Piece
- pieceType:PieceType
- color:Color {readOnly}
+ isColorWhite():boolean
+ isColorRed():boolean
+ isTypePawn():boolean
+ isTypeKing():boolean
+ isTypeQueen():boolean
+ pawnToKing()
+ kingToQueen()
+ toString():String
0..24
plays with
< uses
Input
+ read(input:String):List<Position>

Figure 11: Big class diagram without Launcher class