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 2

Exercise 1:

(1) A redesign we did to remove the getters and setters was to redesign the Piece.getPieceType() method. The getPieceType() method was used to get the PieceType and then check if it is a King or Pawn. Therefore the redesign was done so that two new methods were created (isTypeKing and isTypePawn). These two methods give back a boolean value, and with this value, the checks can be done. Furthermore, two methods were created to make the program extendable and more readable.

(2) The Second redesign that was conducted was to remove the Piece.setPieceType() method. The main purpose of the setPieceType method is to change a Pawn to a King. Therefore the implementation that was chosen is to create a pieceToKing() method. This method then only works for pieces that can be set to a King and not Vice-Versa. (*Later on, this was extended with a kingToQueen method due to our extension in ex 2.*)

(3) Another redesign was to remove the getColor. This was replaced in the Piece class by two methods: isColorWhite() and isColorRed(). These two methods were chosen, as this is mainly the interesting part of knowing. Additionally, to make it easier to compare to variables of Type Color, two methods called getPieceColor() and getCurrentPlayerColor() were added to the validator and game class. These methods have a private scope and are used to know the color of a piece by testing its color and then giving this information to other methods inside the same class.

(4) In a further refactoring step, we could remove the setEmpty() method inside the Field class. However, while analyzing the behavior and usage of this method, one noticed that this method was only used inside the same class. Therefore, the other method, calling the setEmpty method, had the right and the data to change this variable directly. Therefore the setEmpty method was not needed and got removed without replacement.

(5) The first getter's case that is useful and necessary to our code is the getFields() method. It returns the array list of all the Fields on the Board. Therefore, it is central to the Validator and Game class, which uses the method to do important work in the code. The first idea was to use an iterator to remove the getter; however, this would increase complexity by a large margin, and there would be several calls to a potential Iterator even in just one method, not to mention all the other calls which would be required. That is why we decided not to negatively impact code performance and readability by keeping the getter in the code. Also, one could argue that it is a caretaker object as we use it as little as possible but still have a way of accessing the array. Additionally, by refactoring and analyzing the code to use it as little as possible, we saw that the length of the 2D array is taken several times. Therefore we wanted to reduce the overuse of this method and added two getters. We added the getRowLength() and the getColumnLength() methods inside the board. Nevertheless, these are getters that are okay to use, as they are of the immutable type and therefore constant values.

(6) Another refactoring we did was to modify the behavior of the getPosition method inside the Field class. In this case, we did not eliminate the functionality of this method; instead, we

changed the implementation. This method does not return the reference of the mutable type Position. Instead, we are making a copy of the position and returning the copy of the position. This copy is an immutable attribute of integer rows and columns.

(7) The next removed getter is the getField() method which now returns a copy. Because we are working with specific Field objects used in other methods, it was impossible to return a new Field of type Field as we did previously with the getPosition method. So instead, we are returning a new variable pointing to the same field so that it can be used appropriately and correctly by other methods.

(8) GetPieces is another removed getter. It was used in the Player class to gain access to the Pieces and their attributes. However, the same work can be done by simple methods that return booleans or integers. That way, we can still check or ask for information about the Pieces, for example, by asking how big a player's piece assembly is with the playerPiecesSize method, which returns an integer (if it is 0, the game is obviously over).

(9, 10) Lastly, we have found two other cases in which getters are allowed and possible. These two cases are the getters for Row and Column inside the class Position. After studying the code and the usages of these two variables, one noticed that they are set once and then do not change. Therefore we decided to make these variables constants. As they are constants, they are initiated in the constructor and cannot be changed afterward.
Additionally, these two variables are of type integer. Therefore they are of an immutable type. Furthermore, as the variables are constant for the object, these values must be part of the constructor.

Exercise 2:

2.1

Feature: We include an additional piece type: Queen. This includes more strategies and challenges. Here are the rules a queen follows:

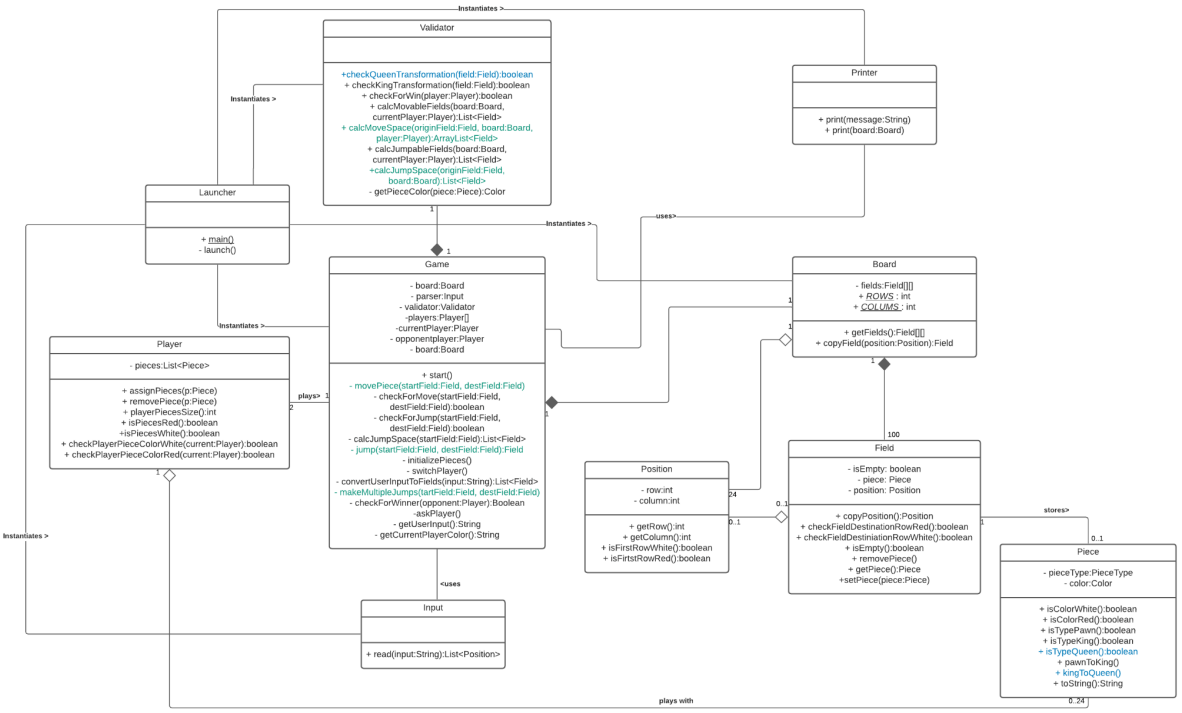
1. A queen must be instantiated from a king. Similar to the change from pawn to the king, when a king reaches the row nearest to the player (first rank) for the first time, the king becomes a queen by a simple move or as completion of a jump. This ends the player's turn.
2. A queen follows the same moving and jumping rules as a king except that a queen can jump over another queen and move (not jump) to an adjacent square on the left or right side and straightforward or straight backward.
3. The jump of a queen can start from both white and dark squares on the board.
4. Other rules are kept unchanged as compared to the standard checkers game.

2.2

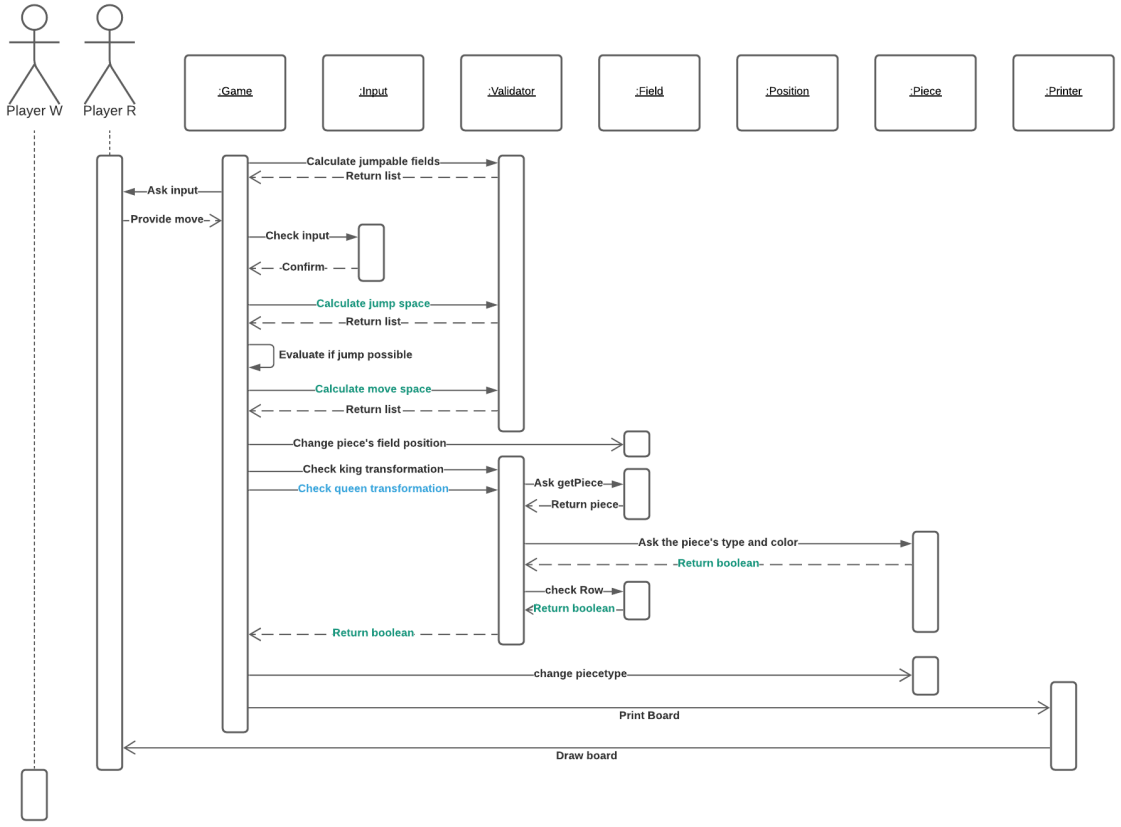
From the perspective of responsibility driven design, we do not need to change the design logic to implement this feature. Therefore, we will not draw the CRC cards here repeatedly as they are kept unchanged. The UML class diagram is shown as follows, the newly added methods are marked as blue, and the modified methods are marked as green - the same also goes for the sequence diagram, which describes the transformation of a king to a

queen. However, the UML class diagram and the sequence diagram are based on the updates made by exercises 1 and 2.

In class Validator(), we update the methods calcMoveSpace() and calcJumpSpace() to let them include the case of queen. We also add the new method checkQueenTransformation() to determine if a king meets the criteria for becoming a queen. We do this check after a move and a jump. When a king can become a queen, we should change the type of this piece and end the turn of the current player. We adjust the methods movePiece() and jump() in class Game() to include this check. Also, the method makeMultipleJumps() is revised to end the player's turn properly.



Picture 1: UML Class Diagram after implementation of the Queen feature



Picture 2: UML Sequence Diagram of a Queen transformation

Exercise 3:

3.1:

So far, our solution contained a Game class that acted as an orchestration class. It asked the player for user input, utilized the parser to process the input, forwarded it to the validator for respective calculations depending on the use case, e.g. jump with a piece or move a piece, executed the corresponding action by updating the board, switched the player after each action, and utilized the printer to print out the board after each action or display the winner. Although the classes the Game depended on, i.e. Board, Parser, Validator, and Printer, were provided by injecting them into the constructor, thus applying the Dependency Inversion Principle by Dependency Injection, the Game depended on concrete classes, such that it was tightly coupled to the concrete classes. So the goal was to reduce the dependencies while loosening up the coupling at the same time.

So far, the Printer was triggered by the Game each time a change was made on the board. In other words, each time a change was made, the Game called the Printer explicitly. On the other hand, the Game had also to call the Board, the Parser, and the Validator at different points of the workflow. On an abstract level, it makes intuitive sense to reduce the responsibility of the Game class and take “calling the Printer” away from it since the Printer

does not contain heavy business logic. Instead of calling the Printer explicitly each time a change on the Board is made, the Printer should listen to the Game and print the Board as soon as a piece is moved. In other words, the Printer should observe the Game for changes on the Board.
Therefore, the Observer Pattern was implemented. It requires the Game to be observable, i.e. to implement an abstract contract IObservable, i.e. an interface, that declares that the Game must implement three methods:

- addObserver(IObserver observer)
- removeObserver(IObserver observer)
- notifyObservers()

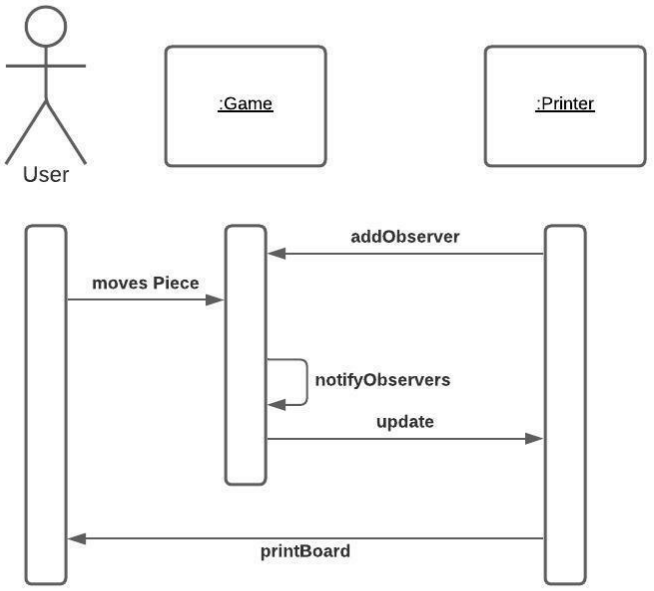
On the other hand, the concrete observer, i.e. the Printer, as it should observe the observable Game must implement the interface IObserver that determines that the Printer must implement the following method:

- update()

With the new setup, it is possible to remove the Game's concrete dependency on the Printer, i.e. removing it from the Game's constructor. In contrast, it stores its observers in a private list and these observers are given to the Game by the Launcher that calls the addObserver method on the Game during initialization. Now each time a change on the board is made, the Game notifies its observers independently from the concrete implementation, i.e. the Printer. It simply iterates over its list of observers and calls on each of them "update()". However, the Game does not know about the concrete implementations anymore since only the abstract type IObserver is stored in its list. However, the Game can rely on the fact that every IObserver must contain an update method to call this method safely.
The concrete observer, i.e. the Printer, must know about the same instance of the Board as the Game. That is fulfilled by applying Dependency Injection during the initialization of the Printer and the Game in the Launcher. That way, the Board instance the Game works on and the Printer prints are the same at all times. With that implementation, the Printer observes the Game, and each time the Game updates its abstract Observers the Printer knows it must update, i.e. print the current Board.

3.2:

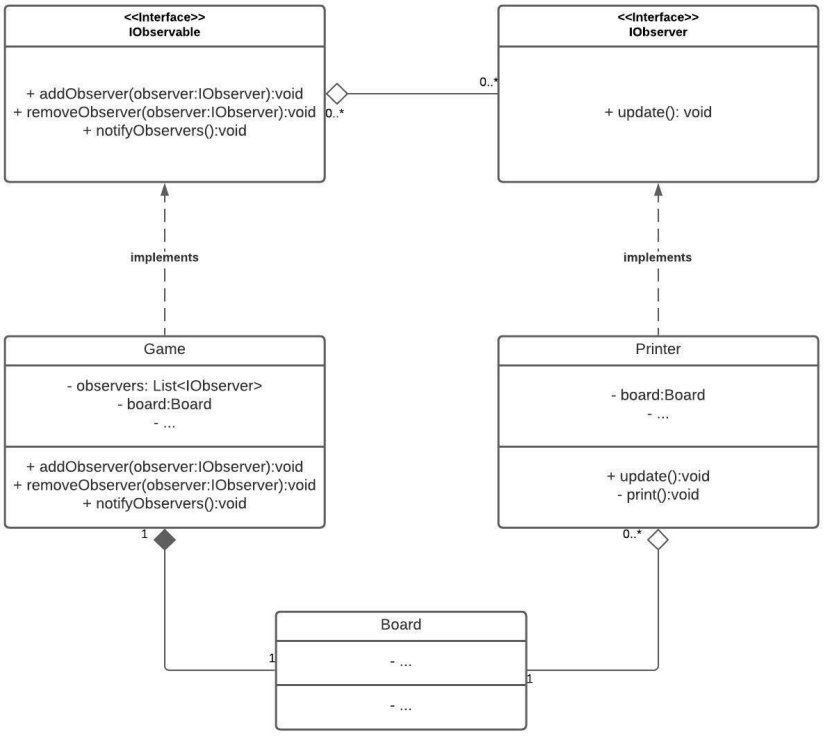
The complete sequence diagram of the user interacting with the game is shown in *Picture 2*. The following sequence diagram assumes the Game has already interacted with the user, validated the input, calculated the possible moves, moved the pieces accordingly, and checked for any transformation. Thus, it focuses on the Game to Printer communication.



Picture 3: UML Sequence Diagram of the Game Printer Communication

3.3:

Again, the following UML diagram omits implementation details of the Game class since the UML class diagram should show the relations between the Game and the Printer. This time, the UML diagram must also include the Board class since it provides the instance that is printed. The launcher is omitted since it simply creates the required dependencies but nothing else.



Picture 4: UML Class Diagram of the Game Printer Communication