Exercise 1

1. Encapsulation aspects from Assignment 1

   When creating a version of Checkers for the first assignment, some aspects of encapsulation were taken into account back then. All the fields were already private and only a few getters and setters in the class `Piece` were present. The class `Board` stored pieces as arrays and never exposed them, the class `Move` stored coordinates and integers privately and only provided them when calling methods of other classes from `Move` itself.

2. Turning public methods to private

   There were numerous public methods that were not actually used from outside of their classes and are not supposed to be used like that. Those methods were changed to private.

   This includes all the methods in the class `Game`, some methods from the class `Move` and some from the class `RuleEvaluator`:
   - `Game.askForInput`
   - `Game.readInput`
   - `Game.getInput`
   - `Game.nextMove`
   - `Move.simpleMove`
   - `Move.jumpMove`
   - `RuleEvaluator.isSimpleMove`
   - `RuleEvaluator.checkForSimpleMove`
   - `RuleEvaluator.checkWinner`

   Objects of the class `Game` are not supposed to be passed anywhere and `Game` is not supposed to provide any interface. Therefore all the methods should be private.

   In regards to the class `Move`, it provides an interface which includes the method `move` which invokes `simpleMove` and `jumpMove` itself. Hence `simpleMove` and `jumpMove` are not supposed to be called from the outside. The class `RuleEvaluator` has a few methods which are helpers for the implementation of other methods in this class and are not part of its interface.

3. Adjusting coordinate variables in the class `Move`

   The coordinates for a move used to be stored as four integers in the classes `Game` and `Move`. Additionally, whenever methods such as `RuleEvaluator.checkValidity` were called, a bunch of primitives were passed. Hence, the code smell primitive obsession was present. The problem was solved by refactoring. The class `Game` no longer stores four primitives and passes them to `RuleEvaluator`, but instead it creates an instance of `Move` directly and `Move` provides an interface of returning coordinates to whoever needs them. Therefore, coordinates are stored only in `Move`. The method `Game.createMove` was added, which creates an instance of `Move` with the input coordinates. Methods in `RuleEvaluator` (`checkValidity`, `isJumpMove`, `isSimpleMove`) were changed to be called with a Move instead of with explicit coordinates. Methods in `Move` were added to access the coordinates. The coordinates are stored as arrays in the class `Move`, and the getters of them provide integers. This is important, as primitives are returned and no reference to array is passed, which could provide an opportunity for undesirable changes. This type of “getter” is necessary, as not only does it improve responsibility driven design (only `Move` is responsible for its coordinates) but also improves encapsulation. Classes are not aware of how coordinates are implemented in `Move`. If implementation changes (arrays to integers or to type of possible class coordinates) - only getters have to be adjusted to still return integers instead of adjusting primitives throughout the whole code as it was initially.

4. Removing getter in the class `RuleEvaluator`
   `getCurrentPlayer`: The only usage of this method outside of the class was to print the `currentPlayer` variable in some messages. This could better be implemented through a `printCurrentPlayer` method. Usage of the getter from the same class was redundant and changed to call on the variable directly.

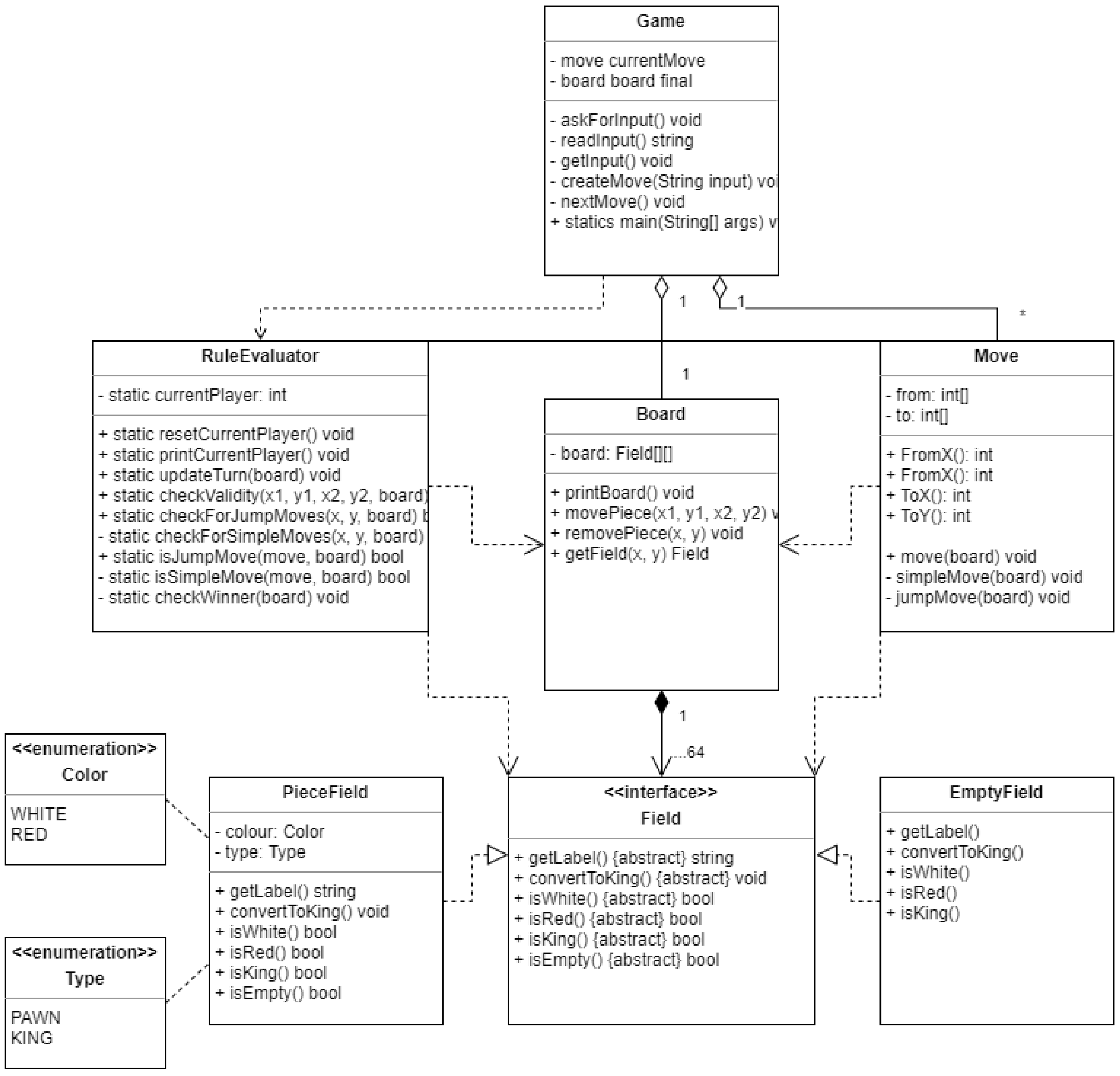
5. Adjusting the `color` and `type` variables in the class `Piece`
   Having these variables be basic characters was an instance of primitive obsession. A better way to represent their values is with enumerations. The public `enum Color {WHITE, RED}` and `enum Type {PAWN, KING}` were added to the class and the datatypes of `color` and `type` were set from `char` to `Color` and `Type` respectively.

6. Removing the getters and setters in the class `Piece`
   `setType`: The usage of this setter was only to set the `Piece.type` variable to `KING`. Since it’s not good to give access to change the `Piece.type` variable openly, the method was rewritten to “`convertToKing`”. `convertToKing` checks whether the `Piece.type` variable is not already `KING`. If it is the method does nothing, if it’s not the player is congratulated and `Piece.type` is changed to `KING`.
   `getLabel`: This method is not a real getter method. It returns a String representation of a `Piece` and is used for printing in `Board.printBoard`.
   `getColor`: This method was used in `Board.isRed` and `Board.isWhite`. It was replaced by an `isRed` and an `isWhite` method in the class `Piece`.
   `getType`: This method was used in `Board.isKing`. It was replaced by an `isKing` method in the class `Piece`.
   `getType`: This method was used in `Board.isKing`. It was replaced by an `isKing` method in the class `Piece`.
   To avoid creating redundant methods in the class `Board` and inducing unnecessary long calls (`move` would call `board.convertToKing`, which in turn would call `Piece.convertToKing`), it was decided to introduce method `getPiece` in `Board`. Strictly speaking, it is not a detrimental getter, because it does not provide opportunity for changes and does not reveal the implementation of class `Board` (It does not provide a pointer to the array `board` and does not reveal that the board is stored as an array). It returns the required `Piece` to whoever needs it. This is

expected as a natural interface of the class `Board`. Since, all the getters and setters were removed from the class `Piece`, whoever gets reference to a `Piece` cannot perform undesired actions.

7. Creation of the interface `Field` which is implemented by `EmptyField` and `PieceField`
   To avoid pre-asking the class `Board` if the field is empty before getting the relevant Piece it was decided to change the class `Piece` into two subclasses `EmptyField` and `PieceField` implementing the interface `Field`. The subclasses implement all the methods such as `isKing`/`isRed`/`isWhite` etc. accordingly and calls to these methods don't have to check first whether a piece is null or not, since `EmptyField` implements all these methods correctly as well. It was decided to have an interface rather than an abstract class because subclasses do not have any common methods. Both `EmptyField` and `PieceField` implement all methods differently. Therefore, the feature of "an abstract class" (providing abstract and non-abstract methods) is not needed. An interface is more appropriate when it provides signatures of methods but without implementation.

The resulting class diagram



Exercise 2

Novel feature idea: Implementing a graphical design of the game.

Requirements

In this game, two users play Checkers using the same computer and input their moves at each round via clicking relevant Fields. When the game starts, it outputs - on the user interface on the screen - the board with the pieces in their initial position.

Board is presented as a collection of 64 black and white Squares in a "chess order". Squares can either be empty or contain pieces. Each piece is represented as a circle-shaped Figure: white or red, with King sign or without depending on whether it is a White KIng/ Red King/ White Pawn / red Pawn.

Each row of the board is indexed by a number (starting from the bottom row with '1') and each column is indexed by a character (starting from the left with 'a'). The game then asks the first player to enter their move by printing the corresponding message in the User interface. For any move one would click the piece one wants to move and click the field one wants to move it towards. These clicks are translated into a Move input.

The validity of the move must be checked; if not valid the user has to enter another move until they enter a valid one. The user should be informed that the Move is invalid and that he touches a wrong Piece by showing the corresponding message in the user interface.

After the player inserted a valid move, the game view is updated and the Player perceives the piece(s) in a new position. Afterwards, the program asks the next user to move, unless the game is finished, in which case it displays a message on the user interface who the winner is.

RDD

1. Going though the requirements and underlining nouns reveals the first set of candidates.

2. Excluding some obvious non-classes:
   - Physical objects: computer, screen -> are not need
   - Other than classes:
     - position or Column & row (probably something else: attribute of Move, Piece e.g.)
     - number and character (input type: int, char)
     - message, chess order, click - > some graphical and interaction elements (probably implemented through GUI)
     - round (probably field)

3. Grouping of relevant candidate classes:
   - Game, game view, user interface -> GUI
   - Board
   - Piece, Field, Empty Field
   - RuleEvaluator
   - Move and move input -> Move
   - Square
   - White King / Red King / White Pawn / Red Pawn
   - Player, user

Going through scenarios and playing CRC cards led to the following design decisions:

1. Player/User again does not get enough of responsibility to form a class

2. Piece and Empty Field 2 subclasses implementing interface Field
3. White King / Red King / White Pawn / Red Pawn extend Square

<u>The final CRC Cards</u>

- **GUI**
  Responsibilities:
    - main (initialize the game, create instance of Board)
    - graphical design of the game (outputting game frame on the screen)
    - interaction with user (showing messages/getting input from clicking)
    - interpreting input and creating moves

  Collaborators:
    - RuleEvaluator (asking it for resetting currentPlayer, getting currentPlayer, checking moves for validity and checking if there are winners)
    - Board (creating it and storing instance of Board and getting fields)
    - Field (getting information about them (isKing/isRed ect.) in order to output correctly)
    - Move (creating Move object and calling method move once valid move is input)
    - Square (using their constructors for printing them out)

- **RuleEvaluator**
  Responsibilities:
    - keeping track that the rules of the game are followed
    - checkValidity
    - checkForJumpMoves
    - checkWinner
    - keep track of currentPlayer and updateTurn
    - keep track of the last move in multiple jump moves

  Collaborators:
    - Board (getting pieces)
    - Field (getting information about type/color/emptiness to perform checks)

- **Move**
  Responsibilities:
    - move (which calls jumpMove / simpleMove)
    - simpleMove
    - jumpMove

  Collaborators:
    - Board (getting Pieces from board, actual moving and removing of pieces - adjusting the Bord)
    - Field (getting information (isKing) to decide when upgrading is relevant in multiple jump move)
    - RuleEvaluator (updating turns after moves, finding out whether the move is a jump move and checking for further jump moves)

- **Board**
  Responsibilities:
    - Providing information about the current state of the board:

        - providing requested Pieces
    - Updating the current state of the board:
        - creating a default board
        - movePiece
        - removePiece

  No collaborators

- **Field** (with subclasses PieceField and EmptyField)
  Responsibilities:
    - Storing and providing information about its type and colour
    - Returning label for printing it (getLabel)
    - Change its type (convertToKing)

  Collaborators:
    - GUI (printing out message when pawn is upgraded)
    - RuleEvaluator (getting the currentPlayer to create clear messages)

- **Square** (with subclasses BlackSquare (with a further subclass) and Empty square):
  Responsibilities:
    - knowing graphical design of squares
    - getting icon and outputting it on itself on the screen

  No collaborators

<u>Design process</u>

1. We familiarised ourselves with the swing library that is commonly used for creating a GUI by following some examples from reliable sources and then extending upon them to create the first playable version of the game with a GUI.

2. As we already had a reasonably well-designed object-oriented program prior to introducing the GUI, we did not have to make massive changes to the already existing classes.

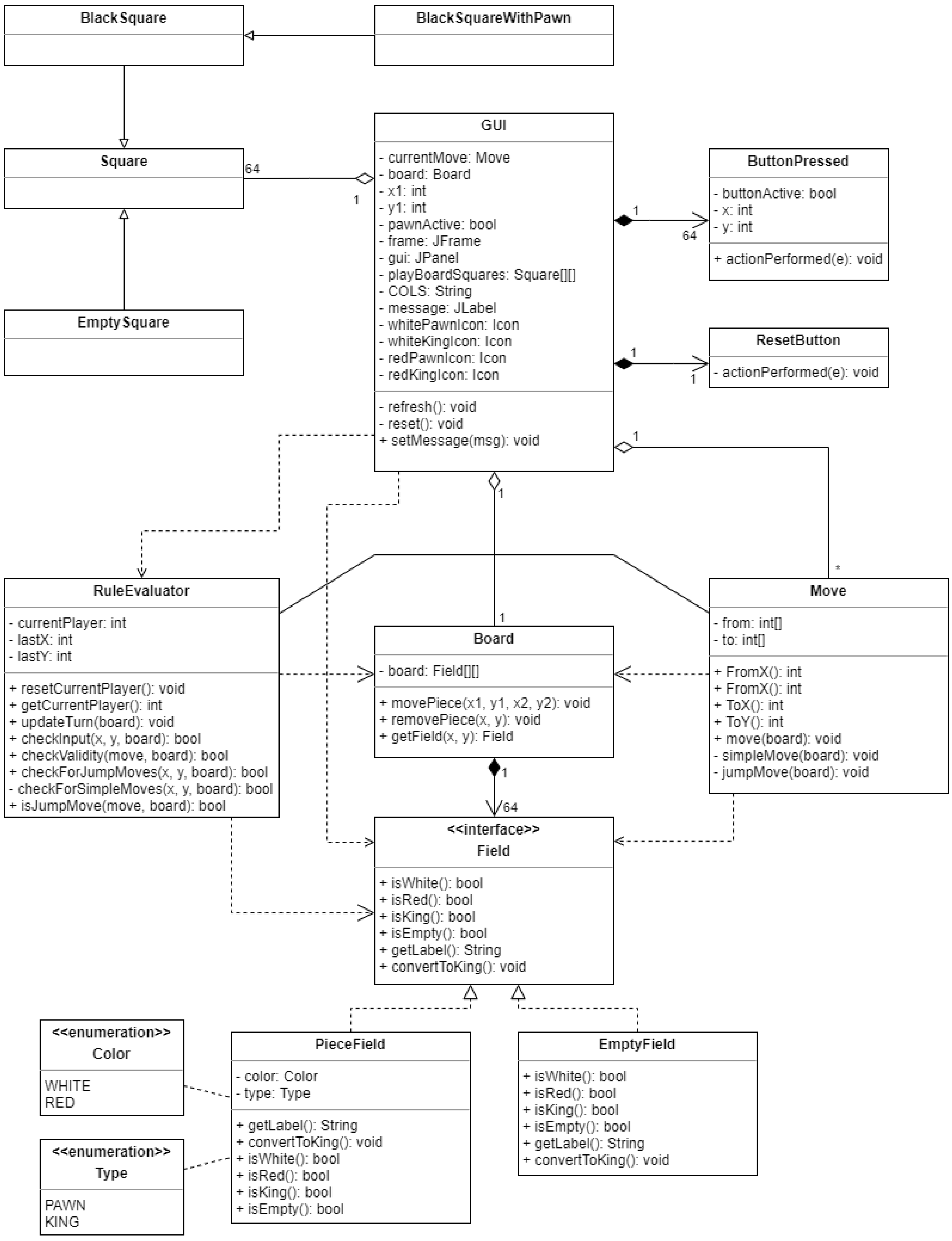
3. As regards the most important changes concerning incorporating GUI into our design, it included:
    - Thinking in terms of responsibilities and collaborations of the classes. The new GUI class would take the responsibility of Game to handle input and the responsibility of Board to print the board.
    - An additional abstract class that would represent a square on the board might prove useful. It could have two subclasses: a white square and a black square which would extend or simply make use of the class JButton in swing. These classes would be responsible for creating the buttons that represent squares on our board based on our template and assigning the functionality to them, this way the GUI class will look cleaner **(essentially extend JButton)**
    - The class GUI will have to collaborate with the class Board to get and store the information about the state of the board as well as with the classes Move and RuleEvaluator. It would, essentially, replace the class Game.

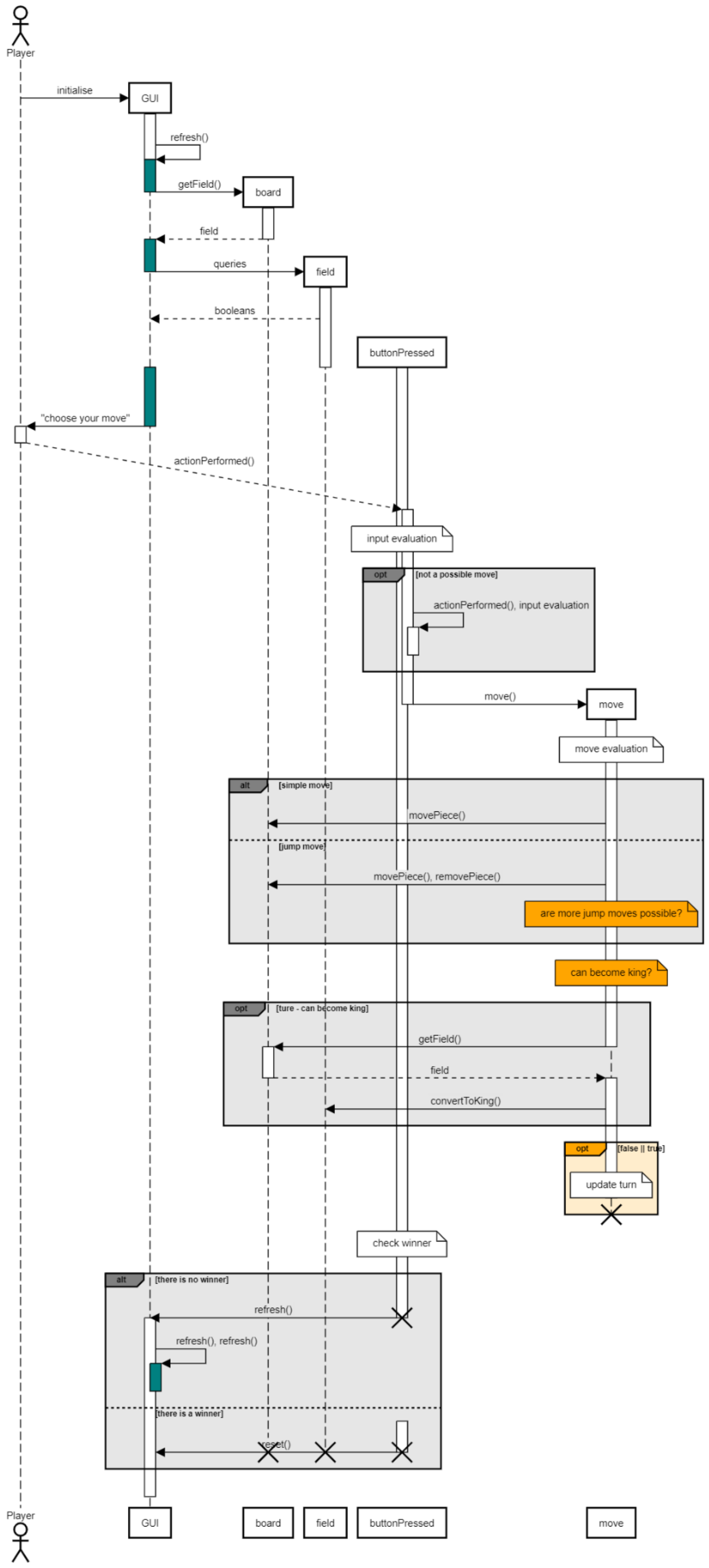
4. It is important to notice that the board in the class Board and the playBoardSquares in the class GUI have very different functionalities and mustn’t be merged. While Board stores the information which is required for the back end of the game, playBoardSquares in GUI is responsible for the front end, so for the visual representation of the board.

5. The getter of the current player for a message in GUI is just fine, as we get the primitive integer and cannot change it. Since it will only be used for printing, the return type is not that important.

6. The GUI allowed us to find two bugs with our rule implementation, since with it we could play through the game more quickly and easily.

The resulting class and sequence diagrams:


BlackSquare
BlackSquareWithPawn
Square
EmptySquare
64
1
GUI
- currentMove: Move
- board: Board
- x1: int
- y1: int
- pawnActive: bool
- frame: JFrame
- gui: JPanel
- playBoardSquares: Square[][]
- COLS: String
- message: JLabel
- whitePawnIcon: Icon
- whiteKingIcon: Icon
- redPawnIcon: Icon
- redKingIcon: Icon
- refresh(): void
- reset(): void
+ setMessage(msg): void
ButtonPressed
- buttonActive: bool
- x: int
- y: int
+ actionPerformed(e): void
1
64
ResetButton
- actionPerformed(e): void
1
1
1
1
*
RuleEvaluator
- currentPlayer: int
- lastX: int
- lastY: int
+ resetCurrentPlayer(): void
+ getCurrentPlayer(): int
+ updateTurn(board): void
+ checkInput(x, y, board): bool
+ checkValidity(move, board): bool
+ checkForJumpMoves(x, y, board): bool
- checkForSimpleMoves(x, y, board): bool
+ isJumpMove(move, board): bool
1
Board
- board: Field[][]
+ movePiece(x1, y1, x2, y2): void
+ removePiece(x, y): void
+ getField(x, y): Field
Move
- from: int[]
- to: int[]
+ FromX(): int
+ FromX(): int
+ ToX(): int
+ ToY(): int
+ move(board): void
- simpleMove(board): void
- jumpMove(board): void
1
64
<<interface>>
Field
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
+ getLabel(): String
+ convertToKing(): void
<<enumeration>>
Color
WHITE
RED
<<enumeration>>
Type
PAWN
KING
PieceField
- color: Color
- type: Type
+ getLabel(): String
+ convertToKing(): void
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
EmptyField
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
+ getLabel(): String
+ convertToKing(): void



Sequence diagram:

Exercise 3

Chosen design pattern: Observer
The previous design suffered from the fact that possible moves were calculated several times per round (when checking validity, when checking for jump moves, when checking for winner). The more logical solution seems to be storing all the possible moves within the pieces and using this information through the whole round. In that case possible moves are calculated only once, the calculation results are stored and reused.

Advantages
the observer design pattern is especially useful because:

1. One-to-many relationship (board -> many pieces)
2. Whenever one object (board) changes -> the rest (pieces) will be automatically updated (so that they can update their moves)

For these purposes the observer fits best. Additional advantages:

3. The Board can register and remove observers. Currently every second field is registered as an observer. If the Board class is ever reused in another game where e.g. each square of the board can be occupied then one can easily add more observers without modifying a lot of the Board code
4. The only thing the Board would know about Field which implements Observer is its Observer interface, particularly the method update(), without knowing implementation details. Technically any other class can start implementing an observer interface and become an observer as well (whoever would like to be updated about changes on board).
5. Board and Pieces can be reused independently, e.g. one can use Chess Pieces or any other objects which have the method update using our Board, in that case Board does not need to be modified.

Implementation

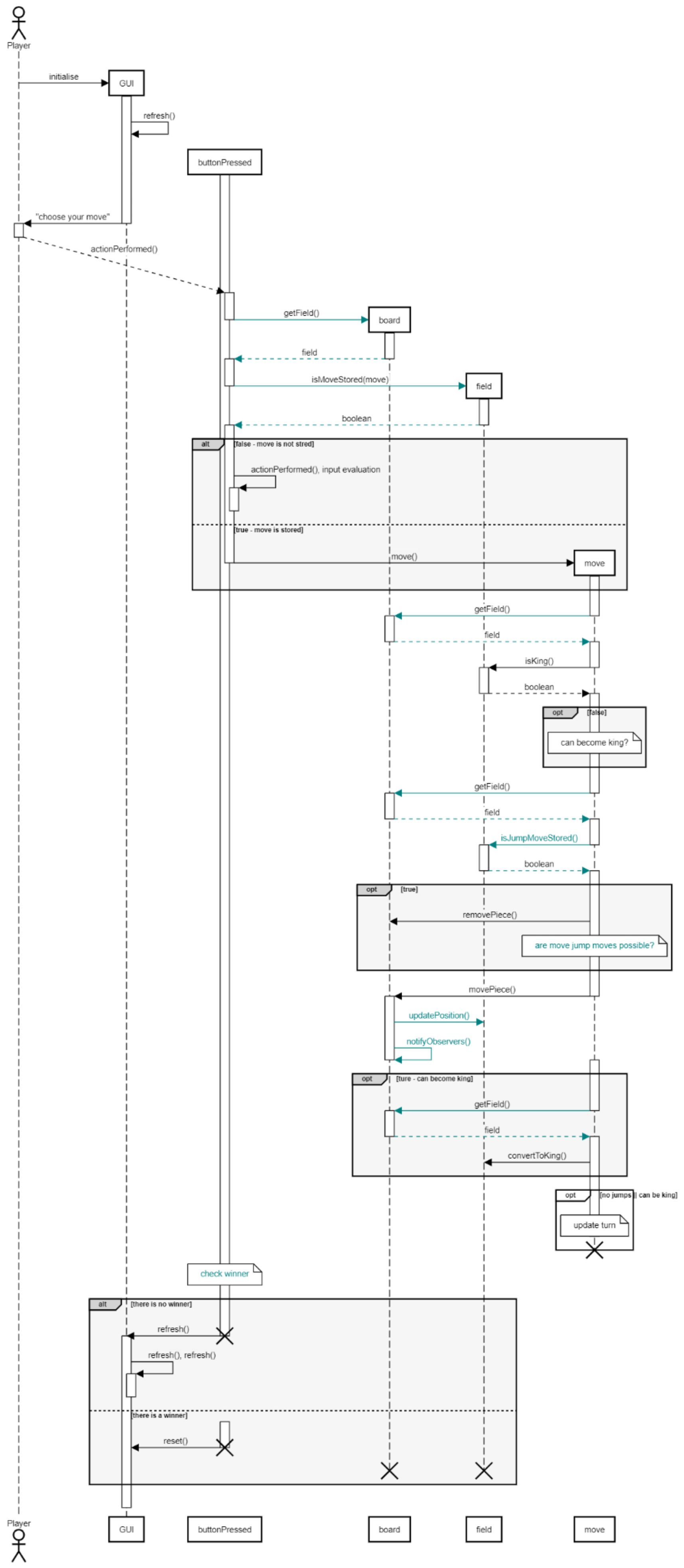
1. Observer interface is implemented by the abstract class Field with its subclasses which has a signature of the update() method. Two concrete observers: PieceField and EmptyField have different implementations of the method update().
2. Board implements interface Subject and hence all its methods. It stores a list of observers, can add some observers, remove them and notify them all.

General idea
Board updates all the pieces whenever it is changed. notifyObservers() calls the update() method inside all the pieces. Every piece stores information about its possible moves. After being updated, they update their possible move. GUI passes moves to the piece in question to evaluate if such a move is possible, the piece returns true or false.

Changes made

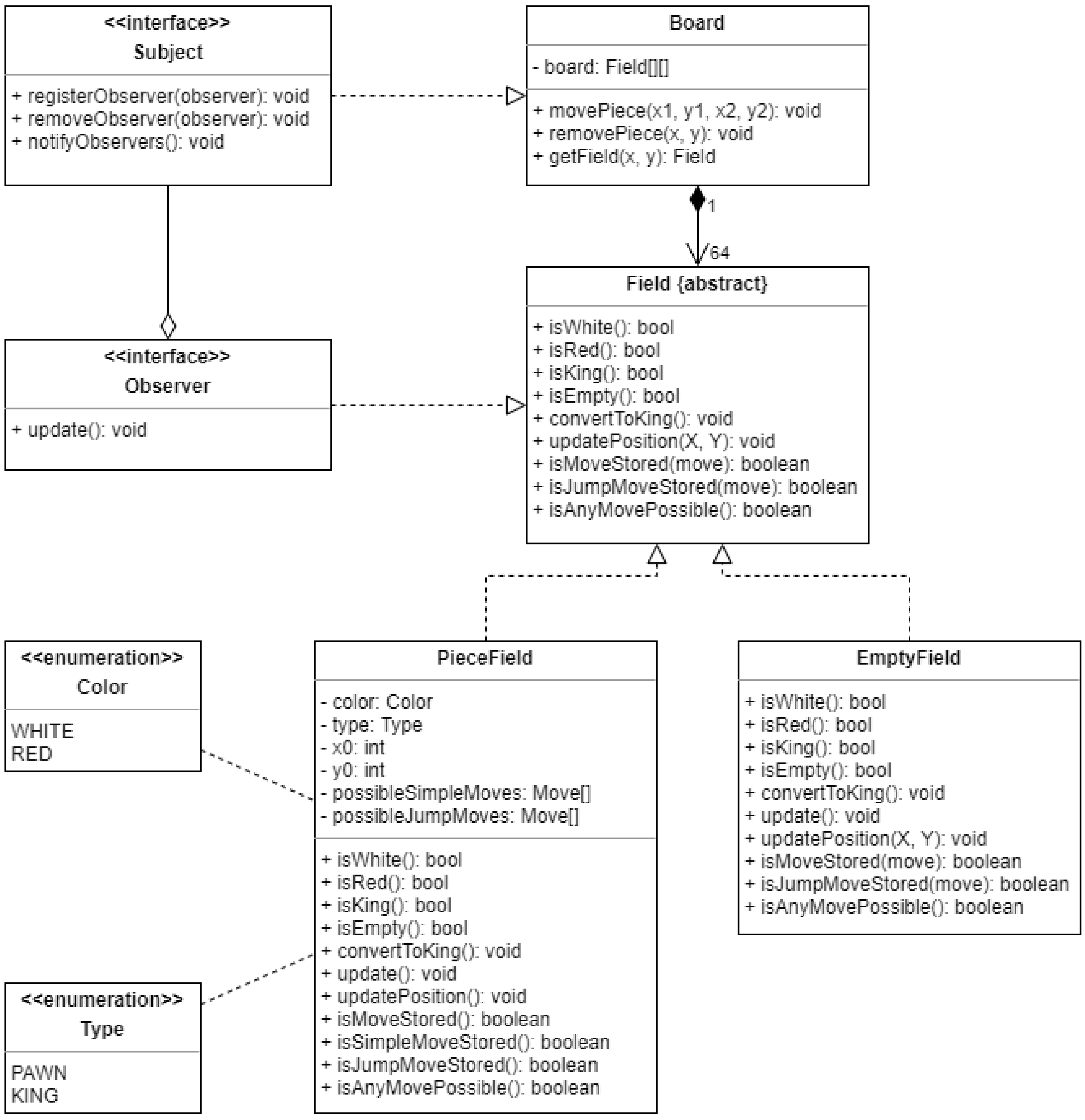
1. Board implements Subject and its three methods: registerObserver, removeObserver and notifyObservers. It stores all observers as an ArrayList. Field implements Observer and its method update().
2. The constructor of the Board is changed to update RuleEvaluator about its new state, update the coordinates in the pieces, register observers and notify them about the current change.
3. Board.movePiece is changed to update RuleEvaluator about its new state, update the coordinates in the pieces and notify them about the current change.
4. RuleEvalutor is changed to have a field of the current state of the Board.
5. Pieces are changed to implement the method update() which invokes updating the arrays of possible moves.
6. Updating of possible moves happens using the static method isValid() from the RuleEvaluator.
7. checkWinner is adjusted by asking the pieces of the current player if any move is possible.
8. GUI and Move are adjusted so that they can ask the pieces directly if a move is valid or if it is a jump move.
9. The method isSimpleMove() is removed from RuleEvaluator.

2. Sequence diagram


Player
initialise
GUI
refresh()
buttonPressed
"choose your move"
actionPerformed()
getField()
board
field
isMoveStored(move)
field
boolean
alt
[false - move is not stred]
actionPerformed(), input evaluation
[true - move is stored]
move()
move
getField()
field
isKing()
boolean
opt
[false]
can become king?
getField()
field
isJumpMoveStored()
boolean
opt
[true]
removePiece()
are move jump moves possible?
movePiece()
updatePosition()
notifyObservers()
opt
[ture - can become king]
getField()
field
convertToKing()
opt
[no jumps || can be king]
update turn
check winner
alt
[there is no winner]
refresh()
refresh(), refresh()
[there is a winner]
reset()
Player
GUI
buttonPressed
board
field
move

3. Class diagram:

Since the purpose is to show how the pattern is implemented, the class diagram below shows only a subset of the full class diagram. It includes interaction between Board and Field which constitute the Observer design pattern. The rest of the system is implemented similarly and can be looked up in the full class diagram in exercise 2. Here, in order to attract the view to the design pattern part we abstract away from the rest of the system.



<<interface>> Subject
+ registerObserver(observer): void
+ removeObserver(observer): void
+ notifyObservers(): void
Board
- board: Field[][]
+ movePiece(x1, y1, x2, y2): void
+ removePiece(x, y): void
+ getField(x, y): Field
1
64
<<interface>> Observer
+ update(): void
Field {abstract}
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
+ convertToKing(): void
+ updatePosition(X, Y): void
+ isMoveStored(move): boolean
+ isJumpMoveStored(move): boolean
+ isAnyMovePossible(): boolean
<<enumeration>> Color
WHITE
RED
PieceField
- color: Color
- type: Type
- x0: int
- y0: int
- possibleSimpleMoves: Move[]
- possibleJumpMoves: Move[]
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
+ convertToKing(): void
+ update(): void
+ updatePosition(): void
+ isMoveStored(): boolean
+ isSimpleMoveStored(): boolean
+ isJumpMoveStored(): boolean
+ isAnyMovePossible(): boolean
EmptyField
+ isWhite(): bool
+ isRed(): bool
+ isKing(): bool
+ isEmpty(): bool
+ convertToKing(): void
+ update(): void
+ updatePosition(X, Y): void
+ isMoveStored(move): boolean
+ isJumpMoveStored(move): boolean
+ isAnyMovePossible(): boolean
<<enumeration>> Type
PAWN
KING