Ex1

State:

<u>Advantages:</u>

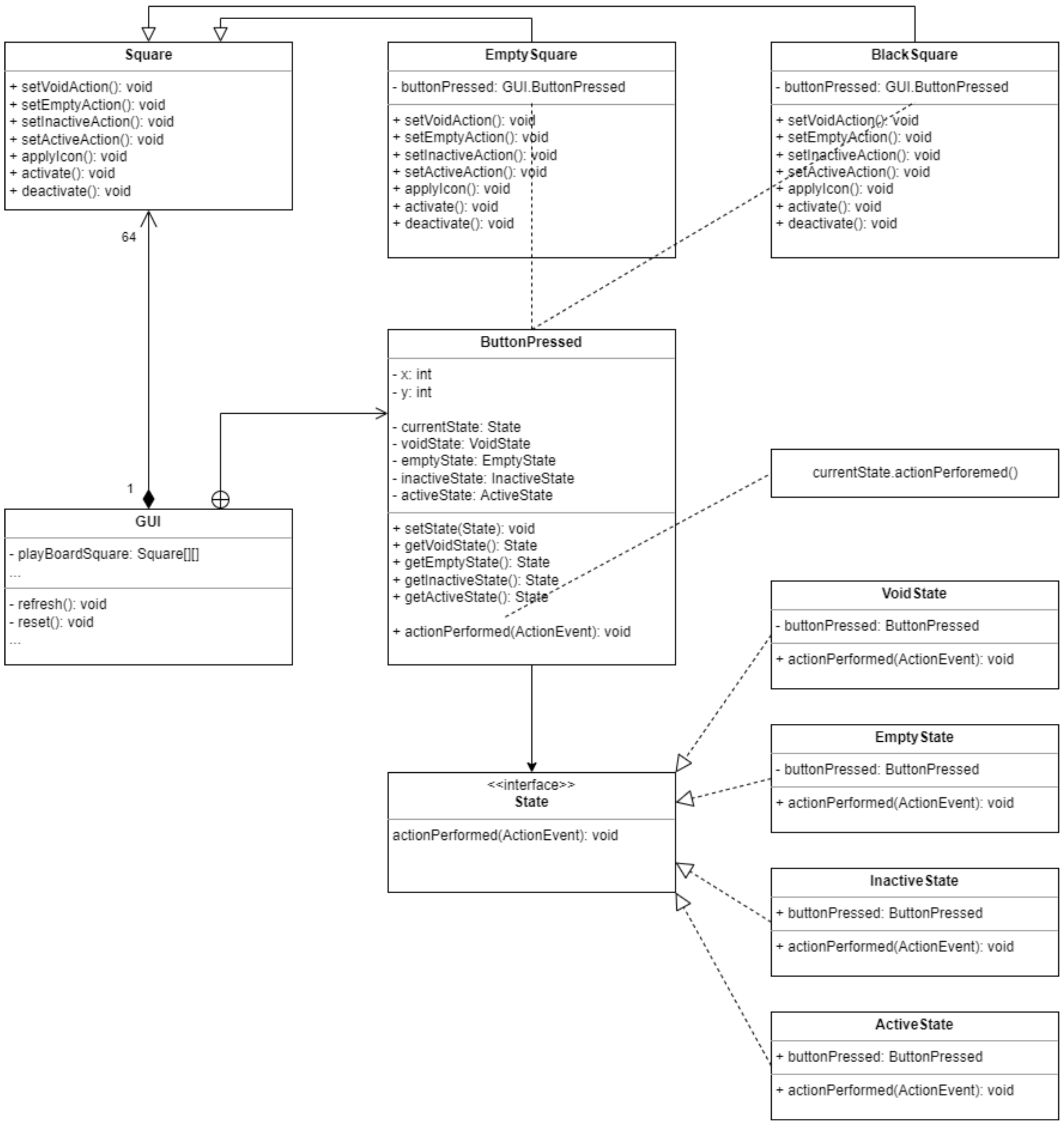
1. The main advantage of the state pattern helps avoid massive nested if-statements, which was exactly the case in the GUI.ButtonPressed class in the actionPerformed method.
2. actionPerformed() in ButtonPressed behaves differently depending on the situation, or state. State dp is specifically designed for such cases when an object is changing its behaviour dynamically.

<u>Implementation:</u>

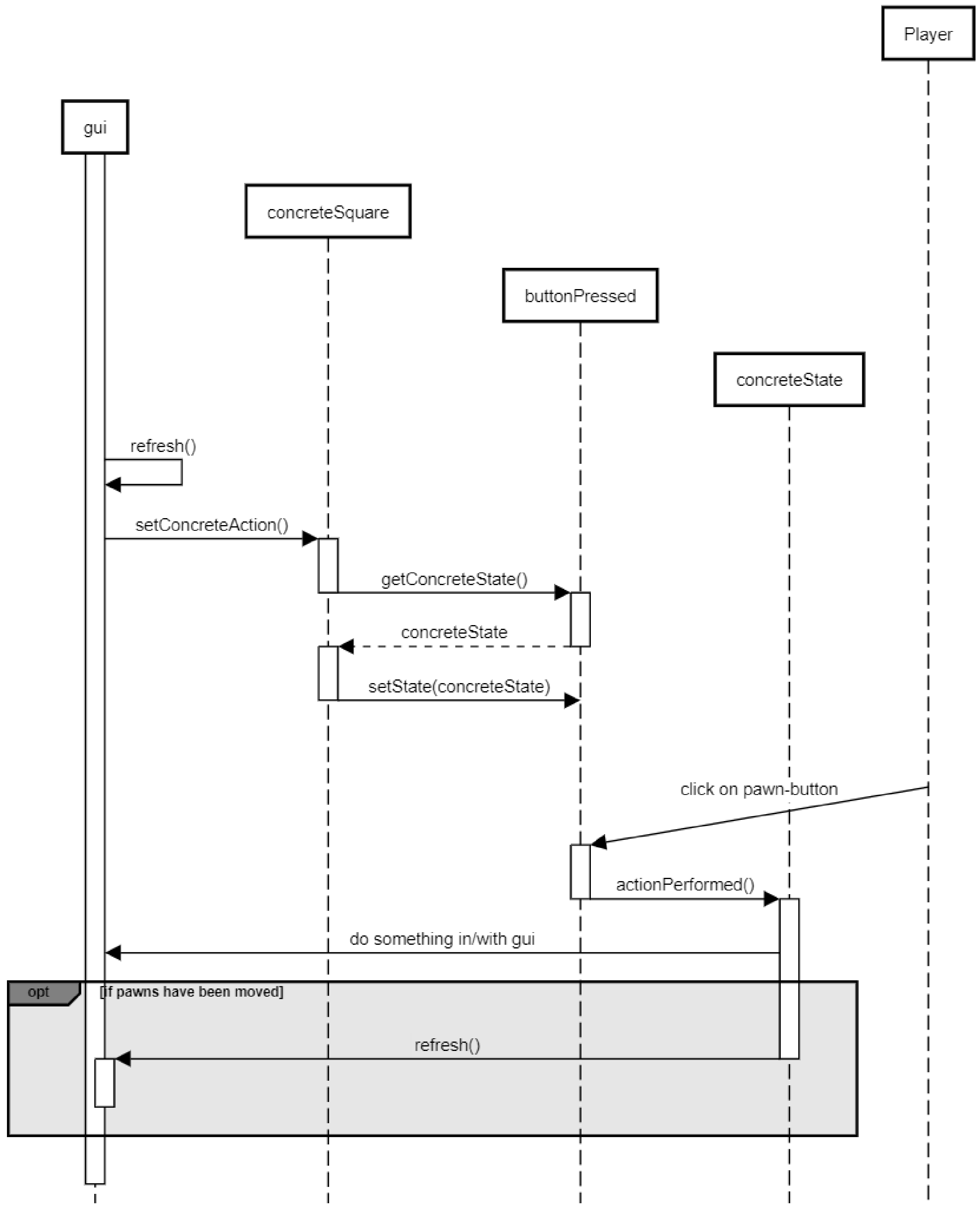
The pattern was implemented in such a way that ButtonPressed has four states that represent different actions a button/pawn needs to perform depending on the situation. Each state is a class which has different implementation of the actionPerformed() method:

- *Void state*: is invoked when a white square is pressed and it throws a message to touch the player's pawn only.
- *Empty state*: when “From” is already selected and the user presses the target square. This invokes the actionPerformed() in this state, which, provided a player has selected their pawn before, checks move validity, implements it, and makes sure to show winners if the game is finished. If the move pressed is invalid an appropriate message is given.
- *Inactive state*: is invoked when the user presses a black square. The ownership over the square is checked and if it corresponds to the current player, the message “please select target field” is given. Otherwise the user message to touch only correct pawns is given.
- *Active state*: in the beginning of the next move. It deactivates the previously introduced moves and shows a message to enter the next move.

All four states implement the interface “State” which includes the signature of the actionPerformed() method. ButtonPressed collaborates with states by getting and setting those and then calling the method actionPerformed().


Square
+ setVoidAction(): void
+ setEmptyAction(): void
+ setInactiveAction(): void
+ setActiveAction(): void
+ applyIcon(): void
+ activate(): void
+ deactivate(): void
EmptySquare
- buttonPressed: GUI.ButtonPressed
+ setVoidAction(): void
+ setEmptyAction(): void
+ setInactiveAction(): void
+ setActiveAction(): void
+ applyIcon(): void
+ activate(): void
+ deactivate(): void
BlackSquare
- buttonPressed: GUI.ButtonPressed
+ setVoidAction(): void
+ setEmptyAction(): void
+ setInactiveAction(): void
+ setActiveAction(): void
+ applyIcon(): void
+ activate(): void
+ deactivate(): void
64
1
ButtonPressed
- x: int
- y: int
- currentState: State
- voidState: VoidState
- emptyState: EmptyState
- inactiveState: InactiveState
- activeState: ActiveState
+ setState(State): void
+ getVoidState(): State
+ getEmptyState(): State
+ getInactiveState(): State
+ getActiveState(): State
+ actionPerformed(ActionEvent): void
GUI
- playBoardSquare: Square[][]
...
- refresh(): void
- reset(): void
...
currentState.actionPerforemed()
Void State
- buttonPressed: ButtonPressed
+ actionPerformed(ActionEvent): void
Empty State
- buttonPressed: ButtonPressed
+ actionPerformed(ActionEvent): void
Inactive State
+ buttonPressed: ButtonPressed
+ actionPerformed(ActionEvent): void
Active State
+ buttonPressed: ButtonPressed
+ actionPerformed(ActionEvent): void
<<interface>>
State
actionPerformed(ActionEvent): void

All the work the previous ButtonPressed used to do is now delegated to the states.



Implementation of the pattern has made it much easier to manage the actions of buttons/pawns as well as introduce changes. For example it has let us allow the player to select one pawn after selecting another without having to deselect the previously selected one first. The feature is difficult to put into words and would have been even more difficult to implement with the massive nested if-statement we had before.

Iterator

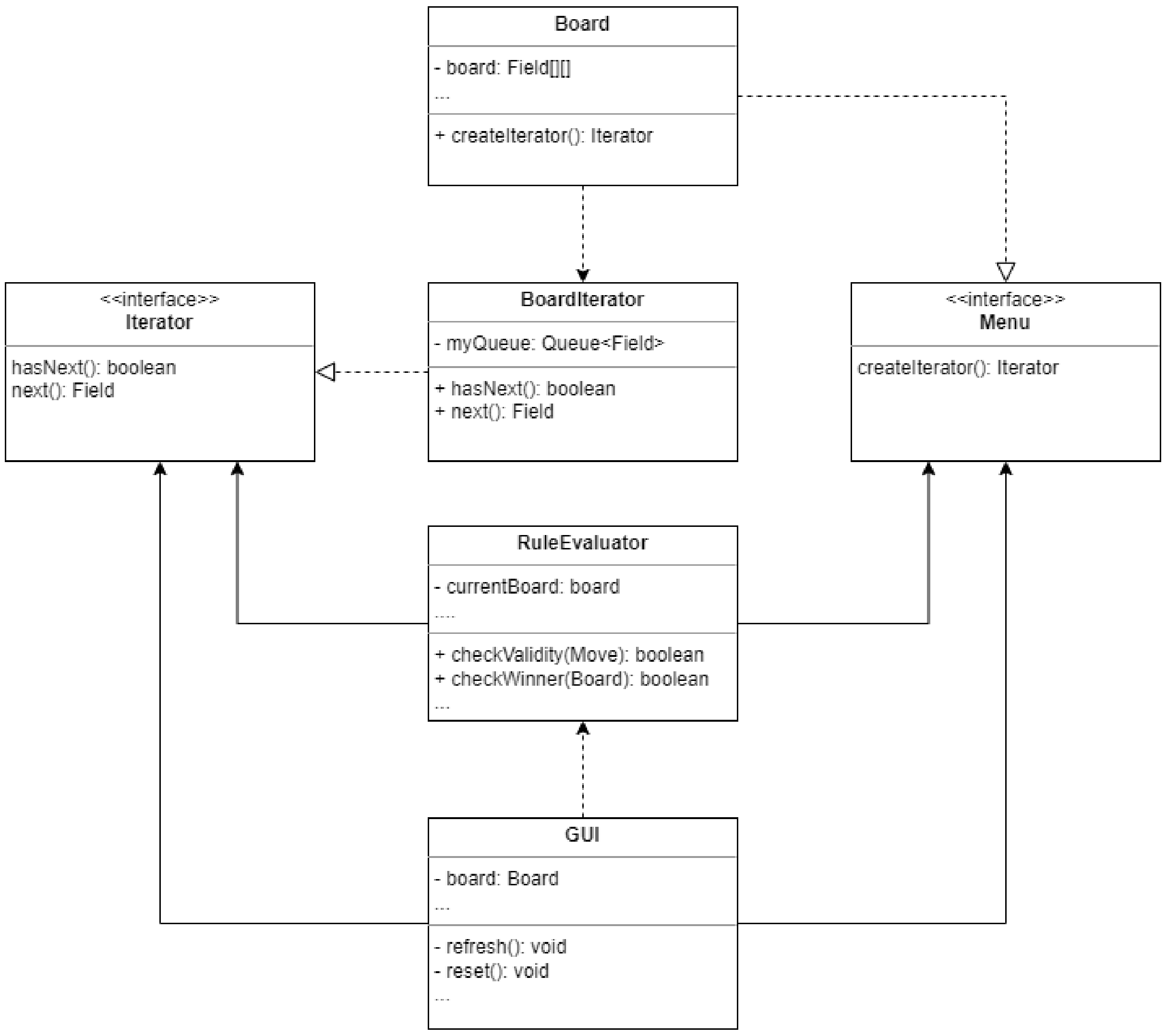
Advantages: Multiple classes (RuleEvaluator and GUI) used to iterate over the Fields in the board of the class Board using nested for-loops. Hence, they had to be aware of the implementation details of the board. The iterator design pattern proofs useful because:

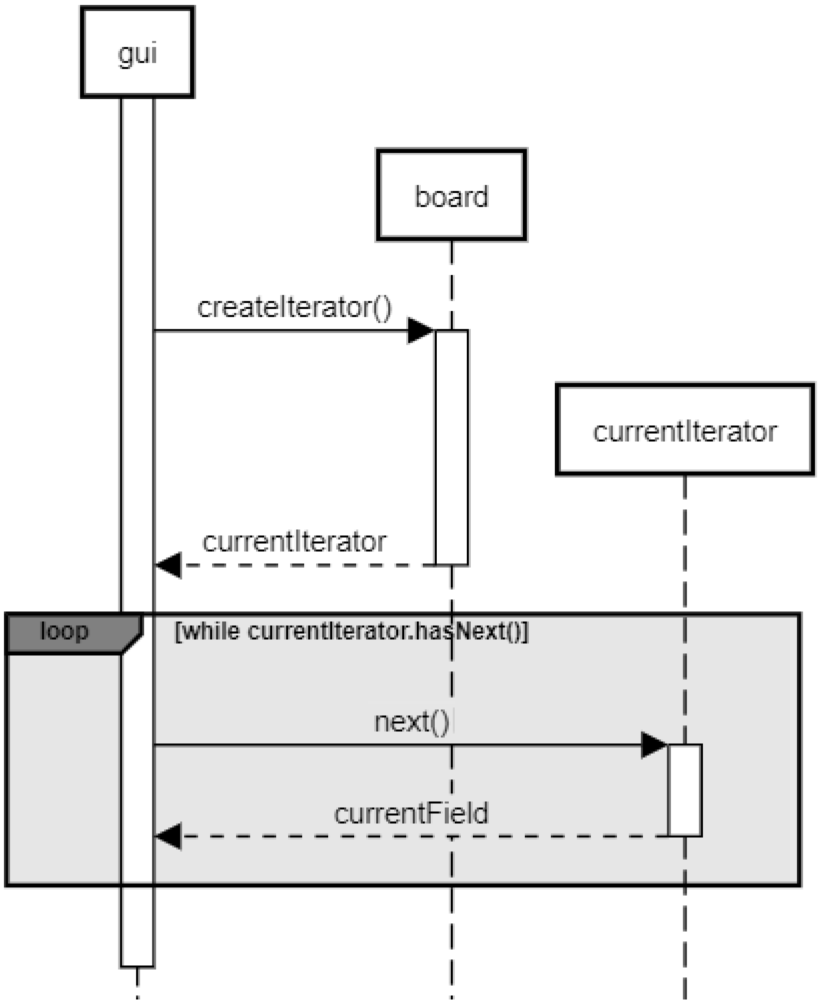
1. It allows encapsulation of the implementation details of the class Board.
2. The repeated nested for-loops in multiple places were removed. The removal of non-trivial iteration makes the code more maintainable and less prone to bugs.
3. If we were to change the storage of the Fields in Board (e.g. increase its size of board), it would have been required to adjust all instances of the nested for-loops previously used for iteration through the Fields. With the addition of the Iterator pattern it is now sufficient to only adjust BoardIterator.

Implementation:

1. The new class BoardIterator was created. It implements the Iterator interface. Queue was chosen as the data structure to store the Fields (stores all the Fields from board, a two-dimensional array). The class implements the methods hasNext() and next() which are available to whoever wants to iterate over the Fields of the board.
2. The new method createIterator() was added to the class Board. It creates an instance of BoardIterator.

Whenever a 3rd-party class wants to iterate over the Fields in Board, it calls Board.createIterator(), which in turn creates an instance of a BoardIterator, which stores all the Fields of the board in a queue. The 3rd-party class then gets access to the methods hasNext() and next().


Board
- board: Field[][]
...
+ createIterator(): Iterator
<<interface>>
Iterator
hasNext(): boolean
next(): Field
BoardIterator
- myQueue: Queue<Field>
+ hasNext(): boolean
+ next(): Field
<<interface>>
Menu
createIterator(): Iterator
RuleEvaluator
- currentBoard: board
....
+ checkValidity(Move): boolean
+ checkWinner(Board): boolean
...
GUI
- board: Board
...
- refresh(): void
- reset(): void
...


gui
board
createIterator()
currentIterator
currentIterator
loop
[while currentIterator.hasNext()]
next()
currentField

Ex2

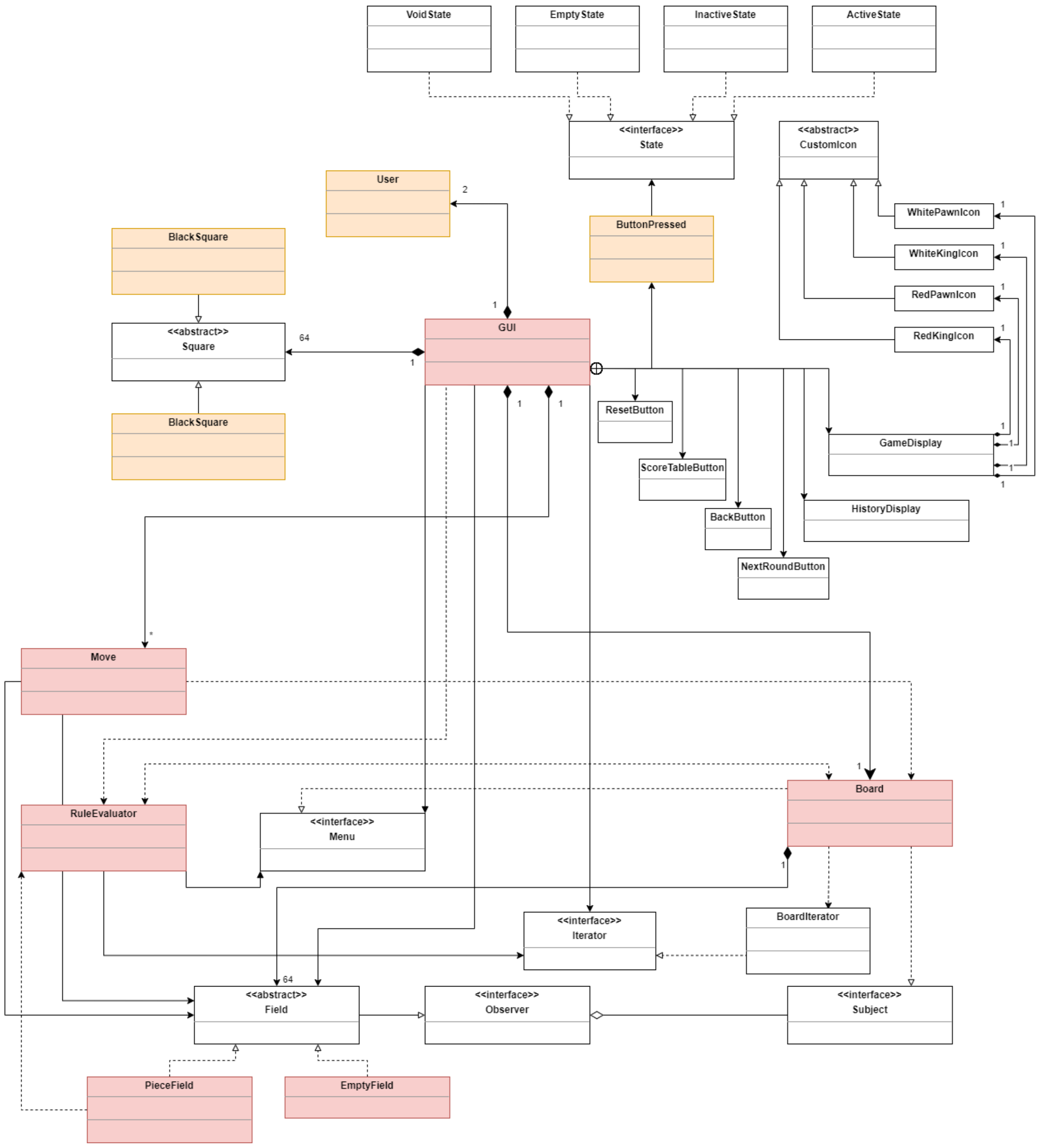
<u>Important classes</u>

- **Board**: The responsibility of the class Board is the current state of the game board. It provides information about its state and updates it when necessary. It is important because:
    1. Many classes rely on Board, such as Move, RuleEvaluator and GUI. So while testing those classes, Board methods will definitely be used and therefore the Board inevitably has to be properly tested.
    2. Board is involved in two design patterns. It implements two interfaces: Subject and Menu. It also has other public methods on which collaborators rely. Hence, it means it is supposed to deliver a significant amount of public methods - the interface of this class is large and it is important that everything works properly.
- **Field (EmptyFiend and PieceField)**: The individual fields of the board are the responsibility of the subclasses EmptyField and PieceField of the class Field. PieceField stores information about the piece it represents. It is important to check both classes because:
    1. Both EmptyField and PieceField have many public methods, meaning that many classes rely on their functionality.
    2. They are involved in design patterns and implement the interface Observer. It is important that the public methods are properly checked.
    3. It is not possible to check other classes without invoking some methods of Field, hence it is inevitable for testing.
    4. Field is involved in the fundamental mechanism of storing relevant moves and updating storages regularly. Such a fundamental mechanism has to be properly checked and tested.
    5. Field is implemented as an abstract class with two subclasses, and they implement an interface. So it is a complicated structure and one needs to make sure that there are no mistakes.
    6. Field is made in a reusable manner, so if reused, one has to ensure the proper functionality.
- **GUI (GameDisplay, HistoryDisplay, ButtonPressed, various state-classes)**: GUI with its inner classes (especially after the changes we introduced in exercise 3) got quite extensive responsibilities. It is important to check it because:
    1. It handles the logic of rounds and game history. Checking other classes would never reveal a problem on that side since only GUI is responsible for it and does not share this responsibility. So if there is a bug there and GUI is left unchecked, testing the rest of the system would not reveal it.
    2. GUI grew quite big with all the inner classes. In the future work we would like to simplify that (move inner classes to outer, ect.) But for now the structure is quite complex and errors could occur.
    3. GUI contains the main method and coordinates the game. Therefore although nobody depends on GUI, any mistake in GUI would lead to incorrect invoking of other classes.

  It must be noted that GUI contains almost no public methods and therefore is not so suitable for unit testing. This time a way to test was found - using reflect. However, the need for GUI to be tested together with its low testability reveals a weak point in our design. A late attempt at refactoring GUI was made, however it was soon realised that sophisticated refactoring would go beyond the scope of this assignment. In addition to that, the only way to bypass the dialogue-box that asks for the users' names at the start of a game would require altering GUI and possibly the experience of playing the game (up to, essentially, removing the feature). It was therefore decided to make the tester press close the dialogue-boxes manually rather than stain GUI with test-related methods/fields and limit its functionality. All the remaining user input is simulated by the tests (various move combinations, to check if GUI works properly).

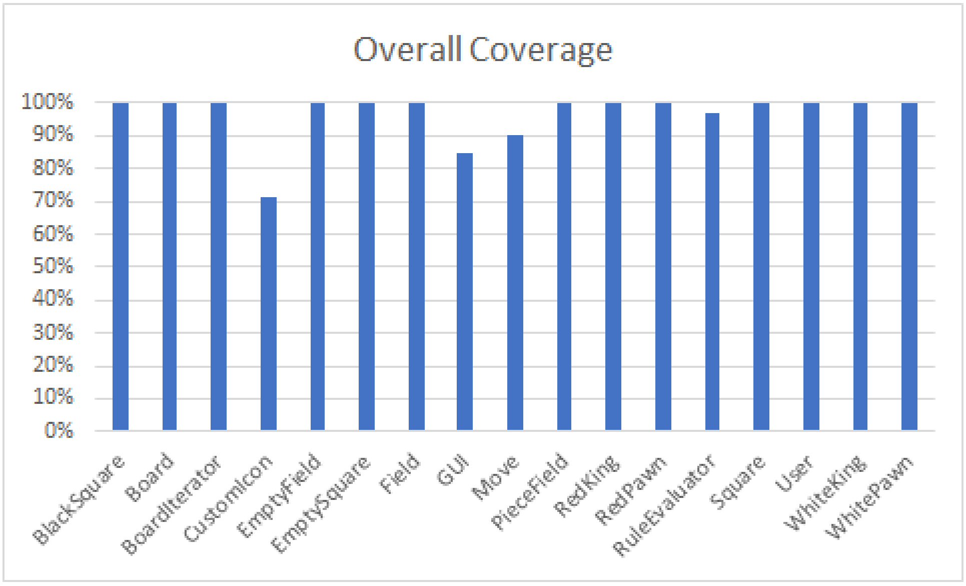
- **Move**: The execution of moves is the responsibility of the class Move. It is important because:
    1. GUI depends on Move. Move has a single but quite important public method.
    2. This public method performs crucial changes on the board and it has to be tested that it does it correctly.
- **RuleEvaluator**: The responsibility of RuleEvaluator is to make sure that the game's rules are followed. It is important because:
    1. Many public methods and many classes rely on it (such as GUI, Board and Field).
    2. It has the important role of handling the logic of the game.
    3. It has a complicated code in checking validity (some if statements, sometimes nested, using the iterator of Board, ect - so there is room for making mistakes).
    4. To check rules of the games manually is very time consuming, e.g. checking for winners would be required to play the whole game. So implementing automated checks of all the rules is essential.
- **BlackSquare and EmptySquare**: Their responsibility is to know the graphical design of all the squares on the board and to output them accordingly. Although they are small and are relatively simple, they do have public methods and GUI depends on them. They can be called important to some extent. That is why they are coloured orange.
- **User**: The responsibility of the class User is to keep track of the user's name and the scores of the game. The User class is very small and with a very straightforward implementation. However, it does have a few public methods and checking of GUI relies on User methods, that's why User is also checked explicitly for completeness. Additionally, the User class is likely to change as the system evolves further and it is important to be able to automatically check it.

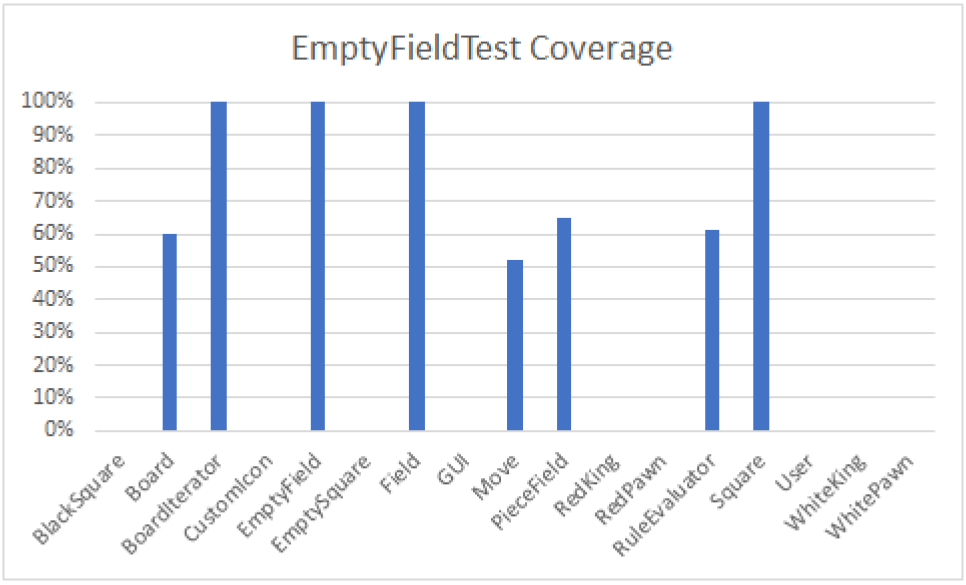
This gets to 10 classes.
Finally, it was decided not to include BoardIterator to classes to be checked, because it closely follows the design pattern and its implementation is unlikely to be changed.

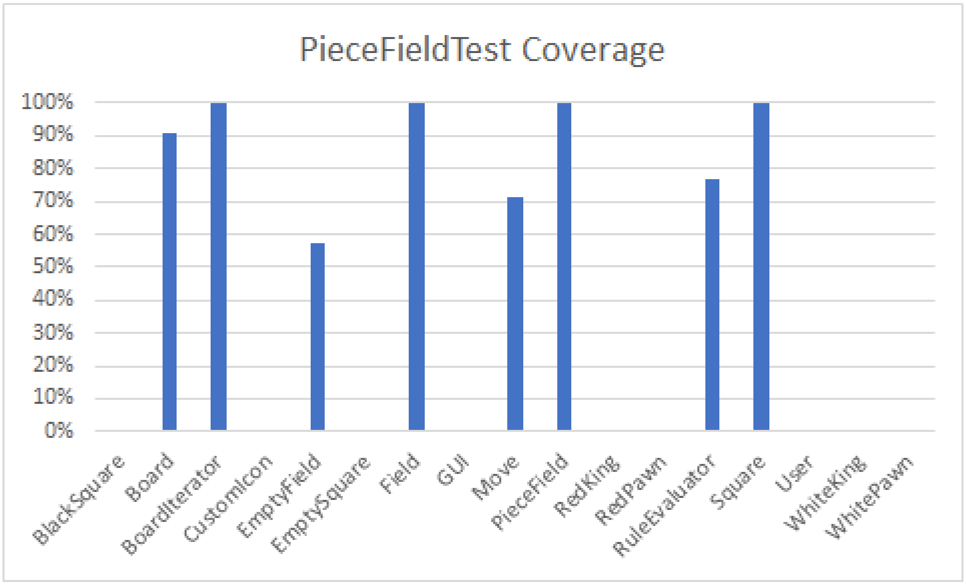


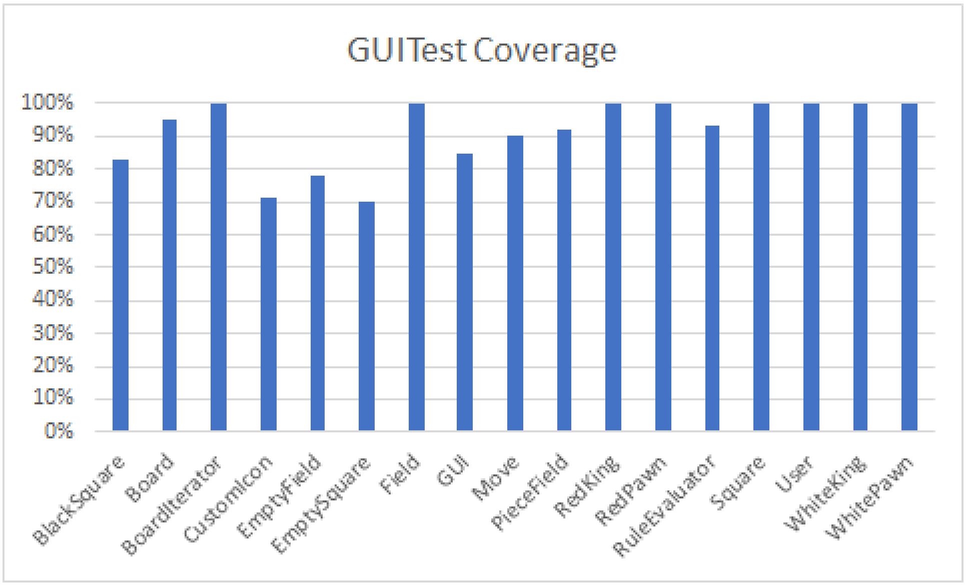
VoidState
EmptyState
InactiveState
ActiveState
<<interface>> State
<<abstract>> CustomIcon
User
2
ButtonPressed
WhitePawnIcon
1
WhiteKingIcon
1
RedPawnIcon
1
RedKingIcon
1
BlackSquare
<<abstract>> Square
64
1
GUI
1
BlackSquare
1
1
ResetButton
ScoreTableButton
BackButton
NextRoundButton
GameDisplay
1
1
1
1
HistoryDisplay
*
Move
1
Board
RuleEvaluator
<<interface>> Menu
1
<<interface>> Iterator
BoardIterator
64
<<abstract>> Field
<<interface>> Observer
<<interface>> Subject
PieceField
EmptyField

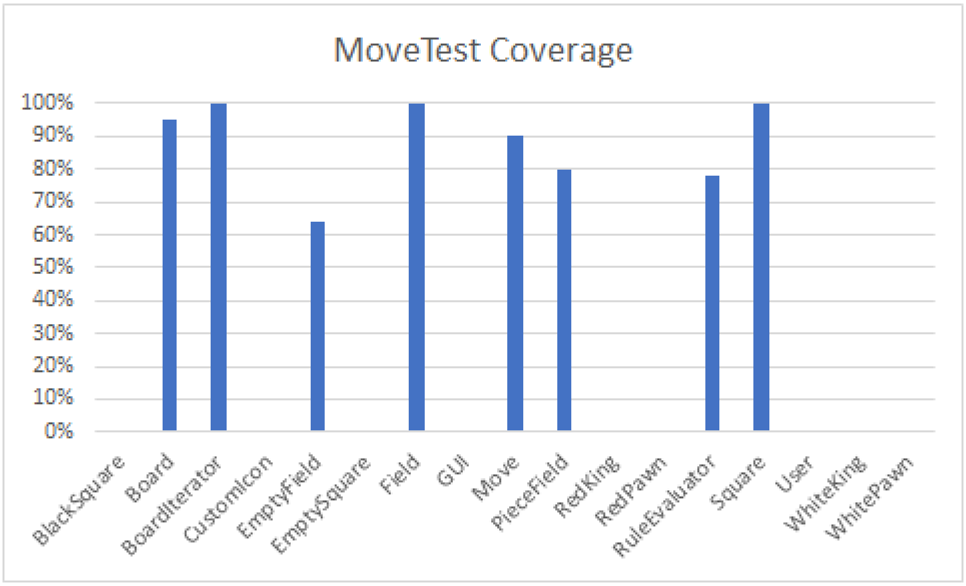
Coverage of tests

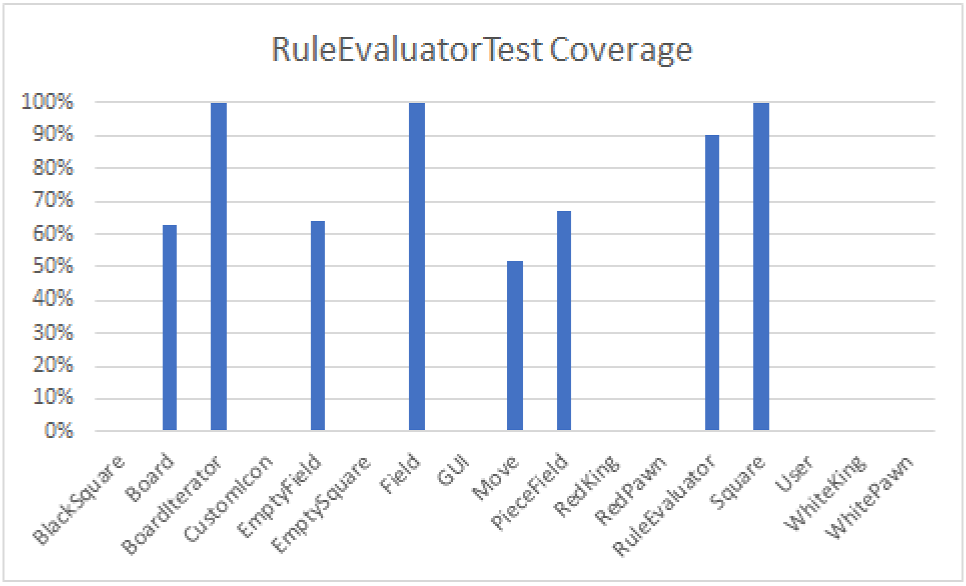


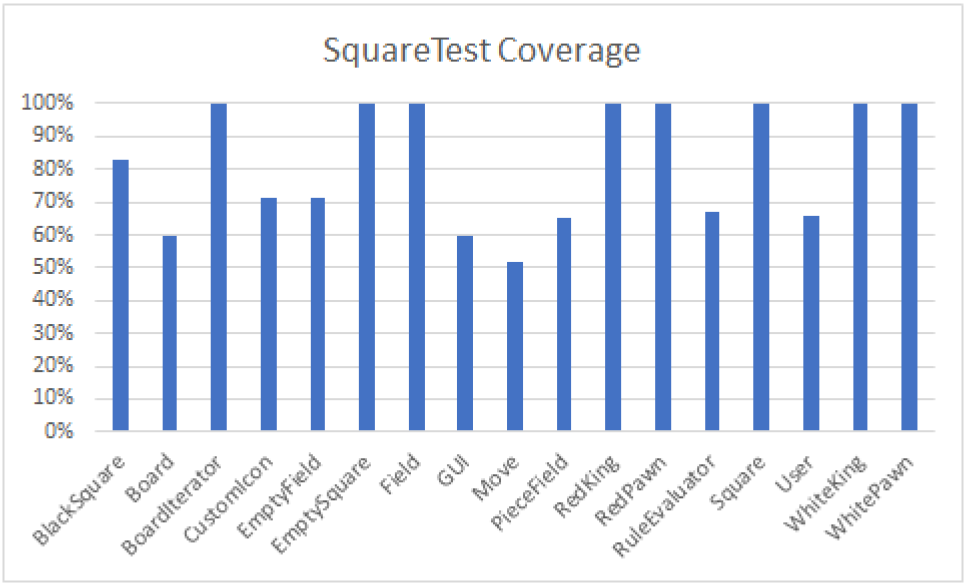


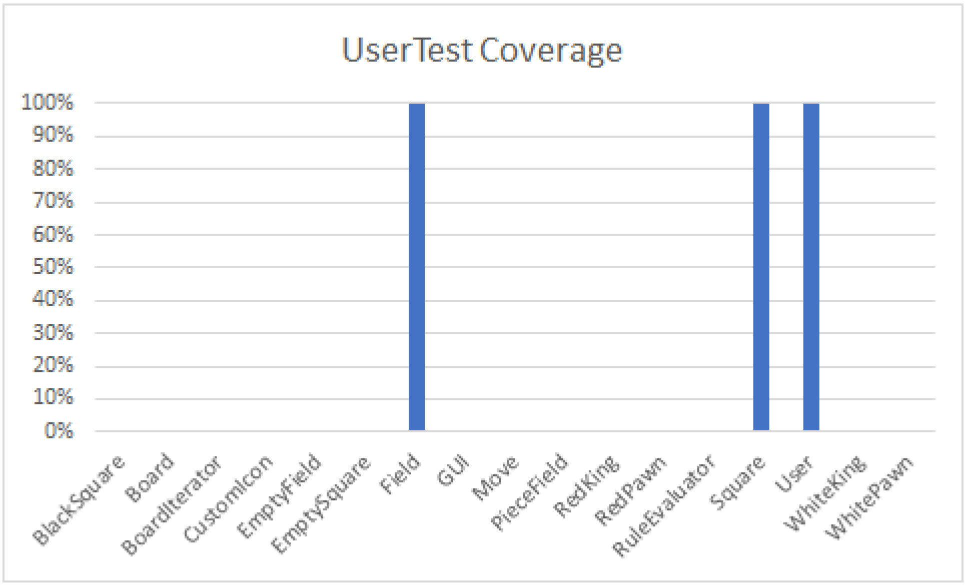



GUITest Coverage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
BlackSquare
Board
BoardIterator
CustomIcon
EmptyField
EmptySquare
Field
GUI
Move
PieceField
RedKing
RedPawn
RuleEvaluator
Square
User
WhiteKing
WhitePawn


MoveTest Coverage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
BlackSquare
Board
BoardIterator
CustomIcon
EmptyField
EmptySquare
Field
GUI
Move
PieceField
RedKing
RedPawn
RuleEvaluator
Square
User
WhiteKing
WhitePawn


RuleEvaluatorTest Coverage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
BlackSquare
Board
BoardIterator
CustomIcon
EmptyField
EmptySquare
Field
GUI
Move
PieceField
RedKing
RedPawn
RuleEvaluator
Square
User
WhiteKing
WhitePawn


SquareTest Coverage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
BlackSquare
Board
BoardIterator
CustomIcon
EmptyField
EmptySquare
Field
GUI
Move
PieceField
RedKing
RedPawn
RuleEvaluator
Square
User
WhiteKing
WhitePawn


UserTest Coverage
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
BlackSquare
Board
BoardIterator
CustomIcon
EmptyField
EmptySquare
Field
GUI
Move
PieceField
RedKing
RedPawn
RuleEvaluator
Square
User
WhiteKing
WhitePawn

Ex 3

Novel feature idea: Allowing for multiple rounds & improving general design.

Requirements (new parts are highlighted in red)

In this game, two users play Checkers using the same computer and input their moves via clicking relevant Fields. The game consists of subgames (called rounds) - as many as the users want. When the game starts, it outputs - on the user interface on the screen - a page, where users are supposed to enter their names and press the button "Start the game". Once this button is pressed the board with the pieces in their initial position is displayed. On the top of the page the current round (initially 1) and messages to the users (calling them by their names) are displayed. The users have two buttons at their disposal: "Reset" and "Check the game history".

When "Reset" is pressed, the users get back to the initial page where they enter their names. When "Check the game history" is pressed, the users can see a table where their scores are given. There a button "Back to game" is displayed. When clicked, the users get back to the display with the board.

Board is presented as a collection of 64 black and white Squares in a "chess order". The 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, a red king, a white pawn or a 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 touched 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 sees the piece(s) on the board in their new position.

Afterwards, the program asks the next player to move, unless the round is finished, in which case it is a page with the table of scores (similar to the one visible by clicking "check the game history"). Here the users are given two buttons: "One more round" and "New Game". "One more round" will start a new round within the same game right away (with the same user names, increasing the round by 1 and updating the score table). "New Game" will reset everything (score table, round, names) and the users will be brought to the initial page.

Additionally, the program will not contain resources (containing pictures of the pieces) to avoid dependency issues. Instead icons are drawn in the programme.

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 needed
    - Other than classes:
        - position or column and row (rather something else: e.g. an attribute of Move or Piece)
        - number and character (input type: int, char)
        - Message, chess order, click -> graphical and interaction elements (implemented through GUI)
        - name, score (attributes of User)
        - table (field)
        - subgames / round (fields in GUI)

3. Grouping of relevant candidate classes (new candidates are in red):
    - game, game view, user interface -> GUI
    - Board and BoardIterator (to iterate over Fields of the board)
    - Field, PieceField, EmptyField (as discovered in previous assignment)
    - RuleEvaluator
    - move and move input -> Move
    - EmptySquare and BlackSquare (as discovered in previous assignment)
    - button clicked / pressed and classes for various states (as was figured out in the previous exercises)
    - player, user
    - Buttons - “One more Round”, “New Game”, “Start the Game”, “Reset”, “Check History” (all in separate classes)
    - Page (separate classes for each class or all in GUI?)
    - Resource, Icon -> CustomIcon

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

1. Player/user gets enough of responsibility to form a class (unlike in previous assignments)
2. For each button a separate class
3. GUI handles round changes
4. Separate classes which extend JPanel - displays: HistoryDisplay and GameDisplay, are created. It was chosen to have separate classes for each of the displays because they would encapsulate functionality of handling JPanel-related things, and GUI would not be aware of their implementation. As it just gets their interface: public methods clear() and update() without knowledge of how they are implemented.
   They were chosen to be inner classes because they share fields and related functionality with GUI. (However after the system was implemented it was noticed that GUI with all its subclasses grew quite large and hence in the next assignment we would consider changing that a bit and prob making a package with just outer classes. For now it would need quite some refactoring and rewriting tests.)
5. Separate classes for resources/icons were made, which just draw themselves.

<u>The final CRC Cards</u>

- **GUI**
  Responsibilities:
    - main (initialize the game)
    - Coordinating the game (start game/round, finish game/round)
    - Proper response to any buttons (creating users when names are inputted, ect.)
    - Graphical design of the game (calling necessary displays to build themselves and puts them to frame)
    - Contains classes - states inside. Each class-state implements the method actionPerformed depending on the state the game is in.

  Collaborators:

    - RuleEvaluator (asking for resetting of currentPlayer, getting currentPlayer)
    - User (creating User, getting its name for outputting on the screen)
    - Board (creating and storing instance of Board, creating BoardIterator)
    - Field (getting information about the Fields in Board (isKing, isRed ect.) in order to output correctly)
    - Square (using constructors for displaying the concreteSquares)
    - ButtonPressed (initialise and attach to the appropriate square)
    - Inner classes (displays - to update display in the frame)

- **GUI.HistoryDisplay() and GUI.GameDisplay():**
  Responsibilities:
- Cleaning and updating itself whenever GUI calls to do so.
  No collaborators

- **GUI.ButtonPressed**
  Responsibilities:
    - Represents the actions taken in the event that a button that represents a Field on the board is pressed
    - Delegates all work to ConcreteStates befitting Context in the state design pattern
    - Stores the relative position of the button it is associated with in relation to the other buttons

  Collaborators:
    - ConcreteStates (stores current state and whenever action performed() has to be called it calls the corresponding current state to perform the functionality. So it delegates all work to the current)

- **GUI.ConcreteStates**
  Responsibilities:
    - Providing implementation of actionPerformed(). Depending on the state, e.g. it can take a move, check and implement it and check for a winner afterwards, or show some error messages whenever the user is doing something invalid

  Collaborators:
    - GUI (updating interface through methods as the game progresses, updating GUI's fields)
    - Board (creating and storing instance of Board and getting Fields by classes-states)
    - 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)
    - RuleEvaluator (for checking for move validity and winners)
    - Square (changing the colour of the associated square through square's methods)

  Responsibilities: Override actionPerformed() by calling the appropriate function in Launcher. E.g. when the user presses reset this button would call Launcher.reset()
  Collaborators: Launcher to delegate it to handle user desired action

- **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 for validity of multiple jump moves

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

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

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

  Collaborators: BoardIterator to create Iterator

- **BoardIterator**
  Responsibilities:
    - Implementing methods hasNext() and next()
    - Storing all the Fields in a queue

  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)

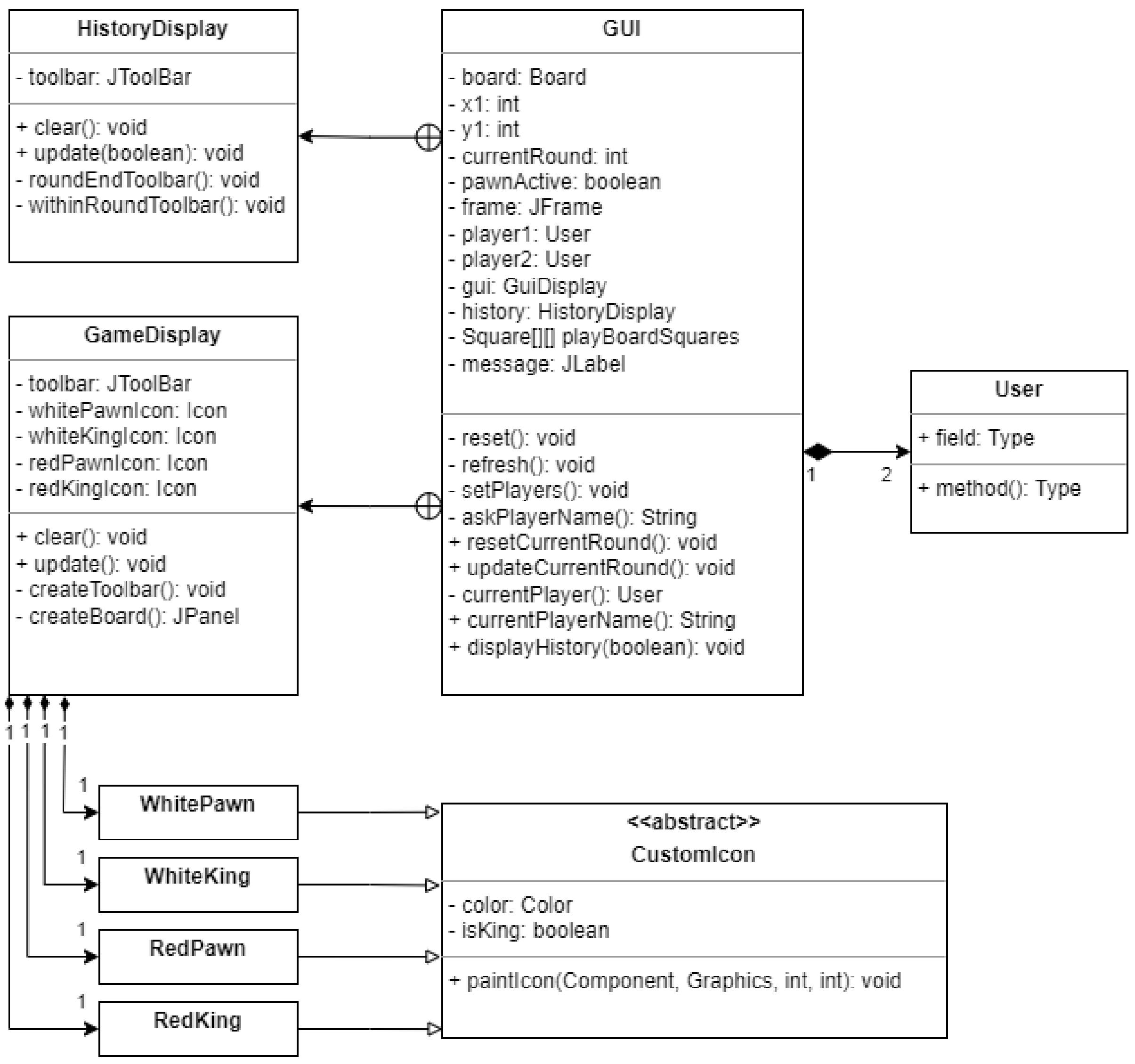
- **BlackSquare/EmptySquare**:
  Responsibilities:
    - knowing graphical design of squares
    - getting icon and outputting it on itself on the screen

  Collaborators:
    - ButtonPressed (buttonPressed represents the action that is performed when the space occupied by BlackSquare/EmptySquare on the gui board is clicked on)

- **CustomIcon(with subclasses RedKing, RedPawn, WhiteKing, WhitePawn)**:
  Responsibilities:
    - knowing how to create an icon of a checkers piece through the paintIcon method depending on the stored fields color: Color and isKing: boolean
    - providing a constructor for the subclasses, which in turn call it in their constructors and provide appropriate values for the fields color and isKing

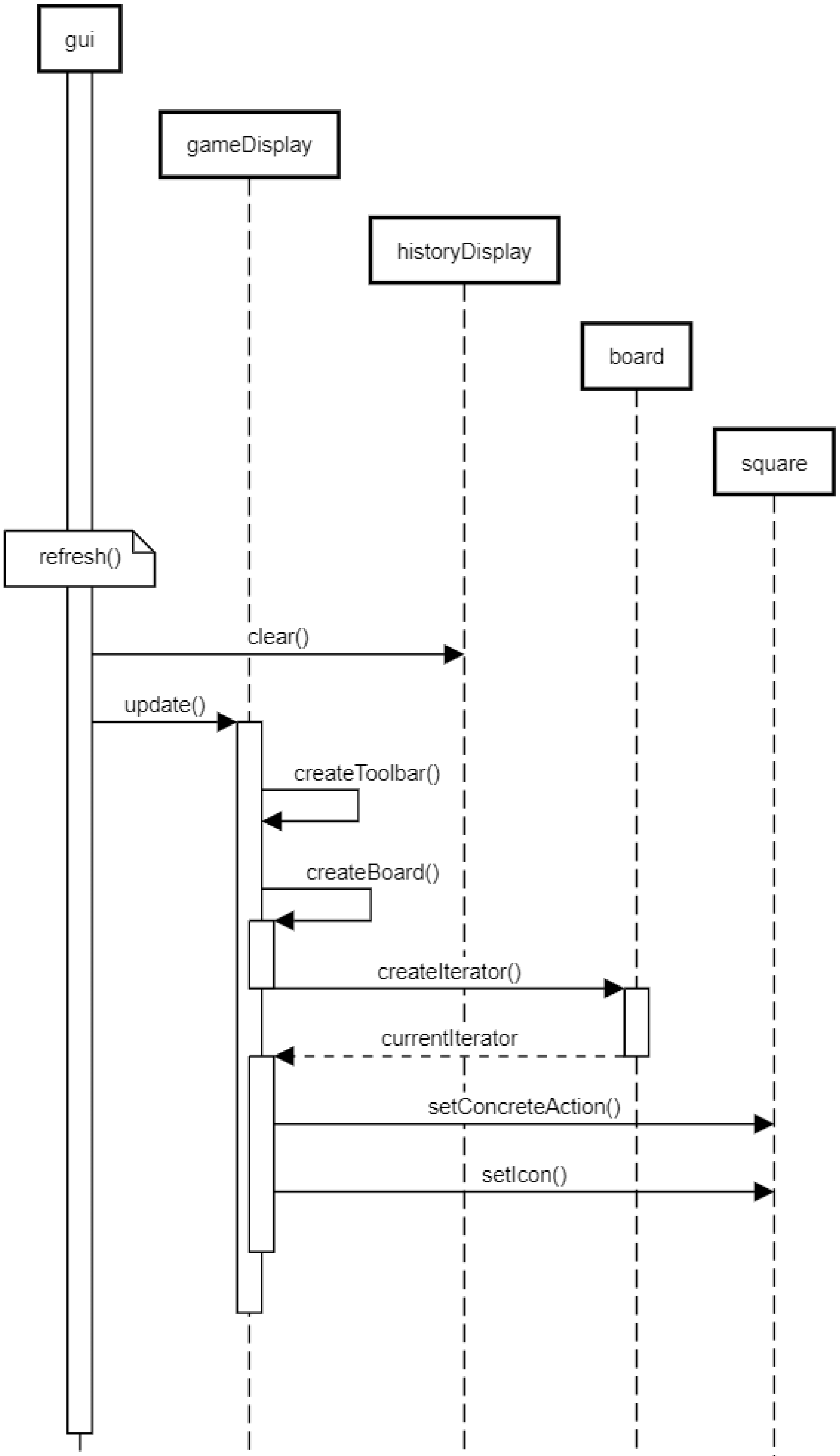
  Collaborators:
    - Square (getting the position and the dimensions of the square in order to be displayed on top of it correctly)

Below we include class and sequence diagrams only reflecting the introduced changes as putting the whole system into one diagram makes it less readable.

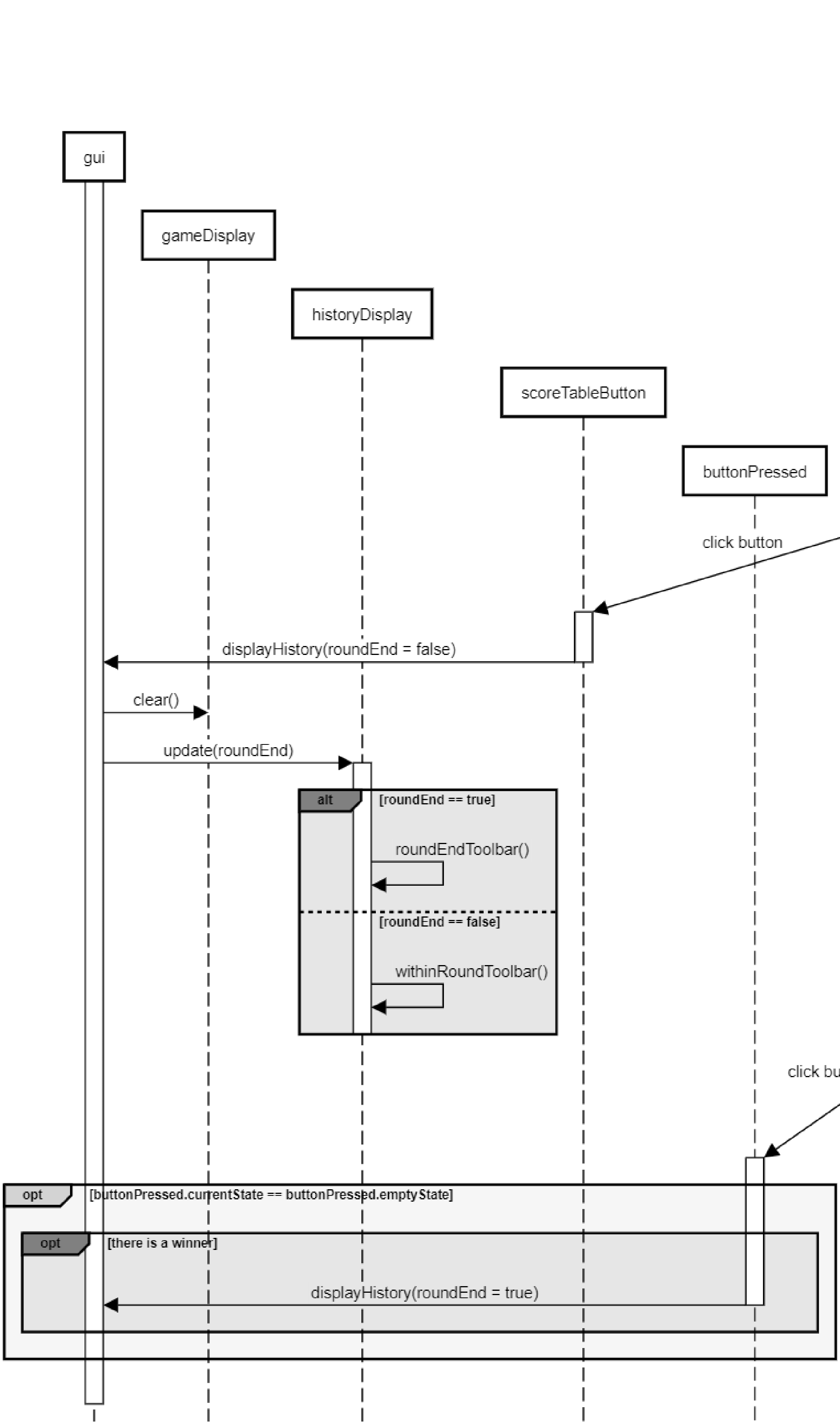


HistoryDisplay
- toolbar: JToolBar
+ clear(): void
+ update(boolean): void
- roundEndToolbar(): void
- withinRoundToolbar(): void
GUI
- board: Board
- x1: int
- y1: int
- currentRound: int
- pawnActive: boolean
- frame: JFrame
- player1: User
- player2: User
- gui: GuiDisplay
- history: HistoryDisplay
- Square[][] playBoardSquares
- message: JLabel
- reset(): void
- refresh(): void
- setPlayers(): void
- askPlayerName(): String
+ resetCurrentRound(): void
+ updateCurrentRound(): void
- currentPlayer(): User
+ currentPlayerName(): String
+ displayHistory(boolean): void
User
+ field: Type
+ method(): Type
1
2
GameDisplay
- toolbar: JToolBar
- whitePawnIcon: Icon
- whiteKingIcon: Icon
- redPawnIcon: Icon
- redKingIcon: Icon
+ clear(): void
+ update(): void
- createToolbar(): void
- createBoard(): JPanel
1 1 1 1
1
WhitePawn
1
WhiteKing
1
RedPawn
1
RedKing
<<abstract>>
CustomIcon
- color: Color
- isKing: boolean
+ paintIcon(Component, Graphics, int, int): void

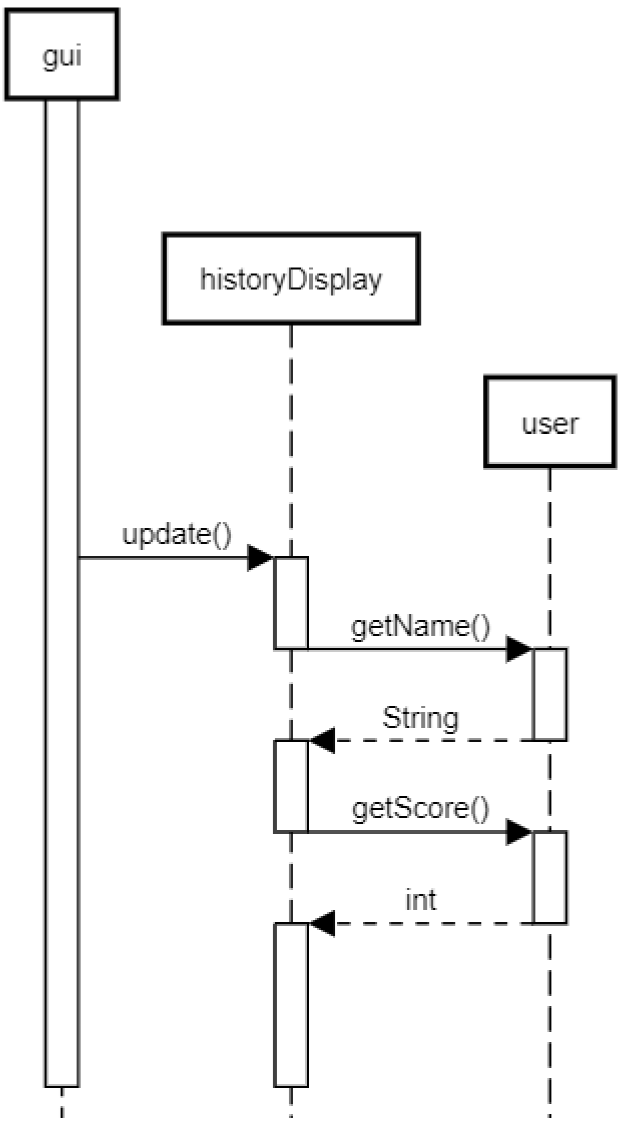
implementation of Displays



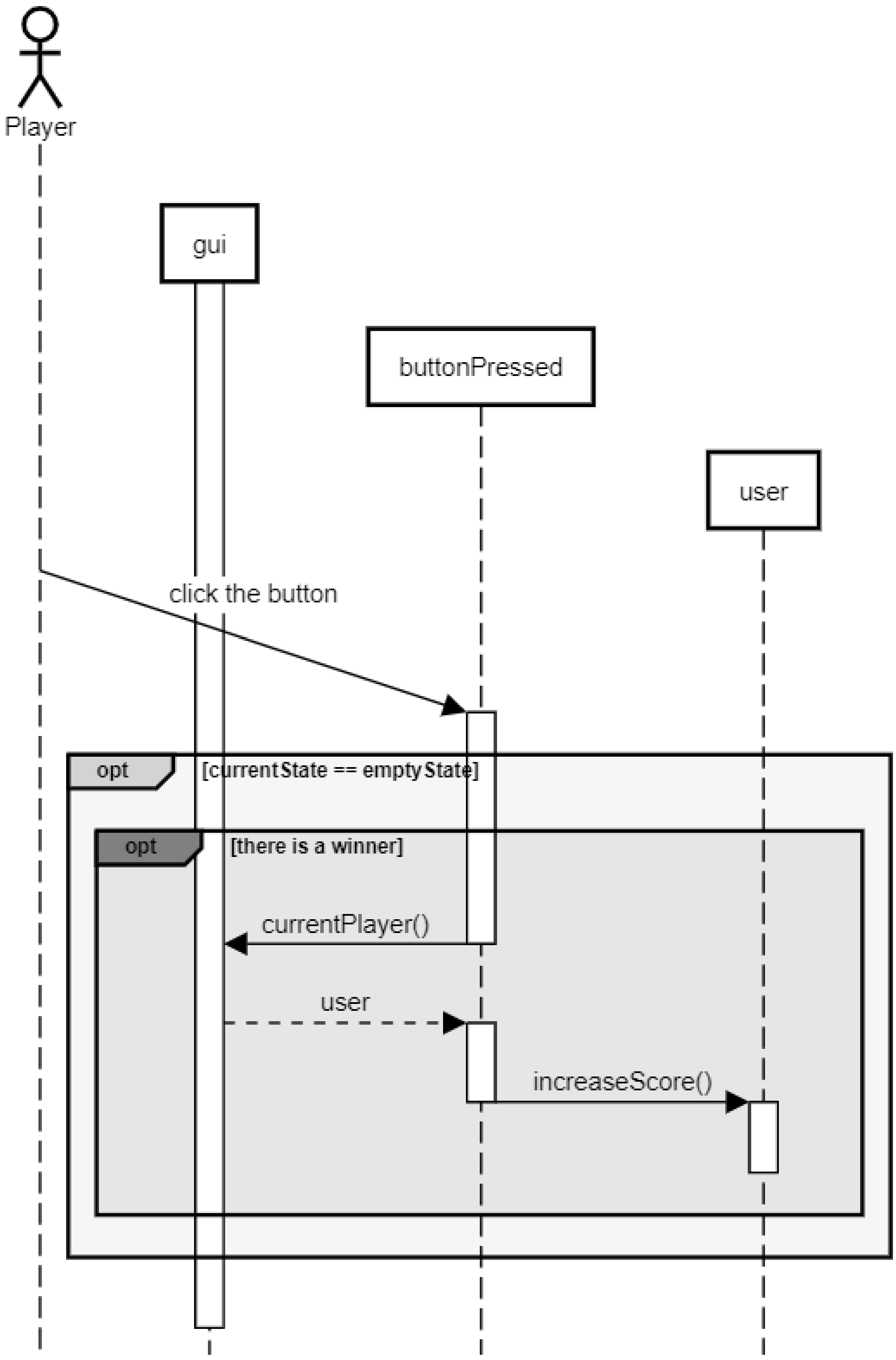
implementation of Displays



implementation of User



implementation of User


Player
gui
buttonPressed
user
click the button
opt
[currentState == emptyState]
opt
[there is a winner]
currentPlayer()
user
increaseScore()