Exercise 1 - A Checkers Game – Two Design Patterns IN! (30 pts)

1. We choose the following 2 design patterns in our Checkers Game code:
    a) State Pattern
    b) Singleton Pattern

    1.1. State Pattern

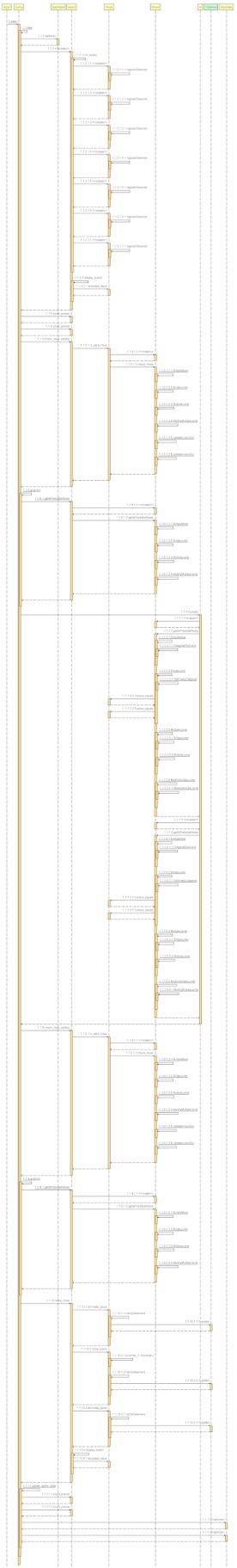
    We have implemented this pattern in order to determine the State of the Game. In order to implement this pattern, we implemented GameState interface, StartState, StopState, and Context classes. In the State pattern the class behavior changes based on its state. That's why it comes under behavior pattern. Context is a class which carries a state. If we wanna add more states in our game in future, it can be done very easily by adding another class of that state. Currently, we just have two states in our game i.e. start state in order to start the game and stop game in order to terminate the game by announcing the winner. This design pattern perfectly fits in the implementation since we already have game_state private class variable in Game class.

    1.2. Singleton Pattern

    We have implemented Singleton design pattern in our AI class. AI class is a new class and is part of Exercise 3 where we have to make some improvements. This class is used to play Single player (play with computer) game mode. Singleton design pattern is a simple yet very powerful design pattern because it is one of the best ways to create an object. It comes under creational pattern. We initially thought of implementing Singleton in Board class but Board class already has an implementation on observer pattern and we wanted to be more creative than using multiple patterns in a single class. Therefore we decided to do it in AI class. Since we wanted to create a single object for AI class, the Singleton design pattern was a perfect fit for it.

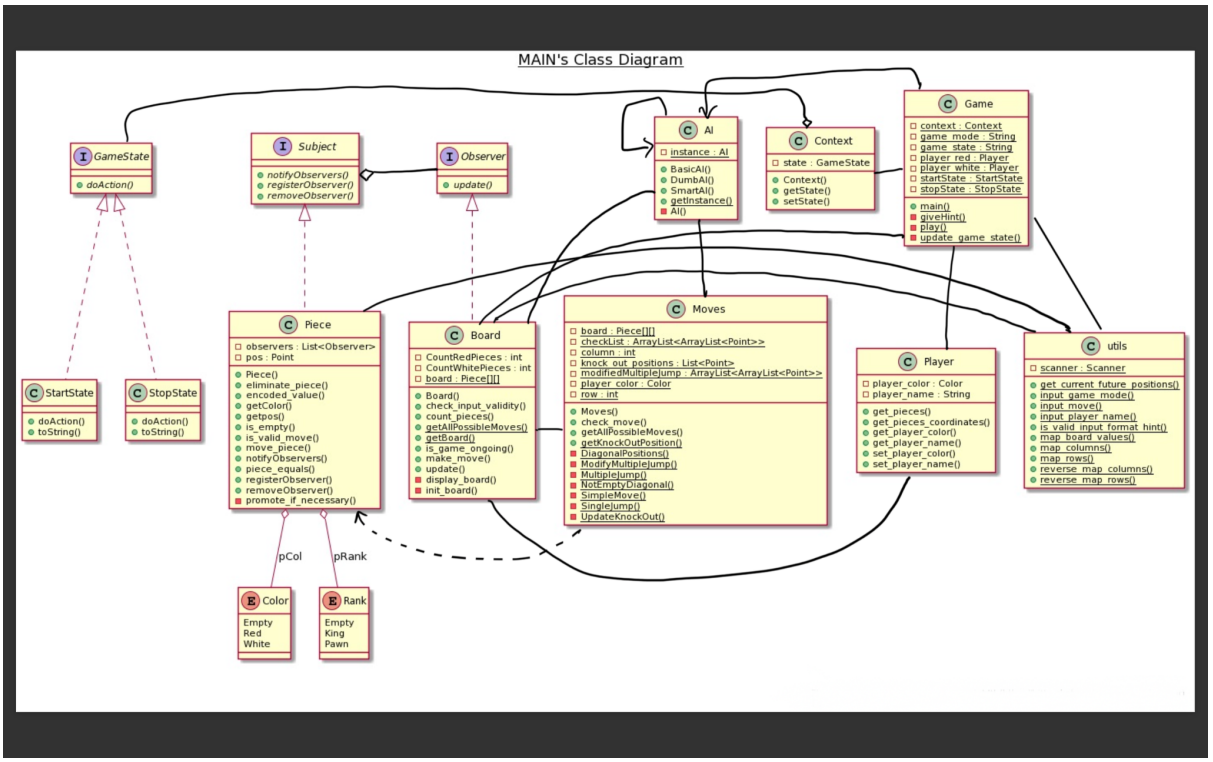
2. **Sequence Diagram**

    The following diagram is just a placeholder since it is not possible to add the full sequence diagram in a way that it is fully visible in the report. So, we have added a high quality PNG image in the root folder of "question_1_3" with the file name Sequence_Diagram.png *(link)*



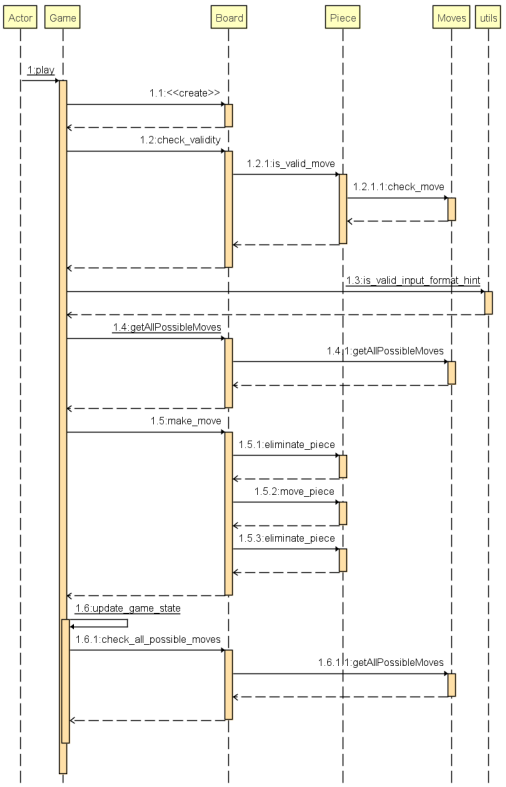
3. **Class Diagram**

Please find the class diagram below:



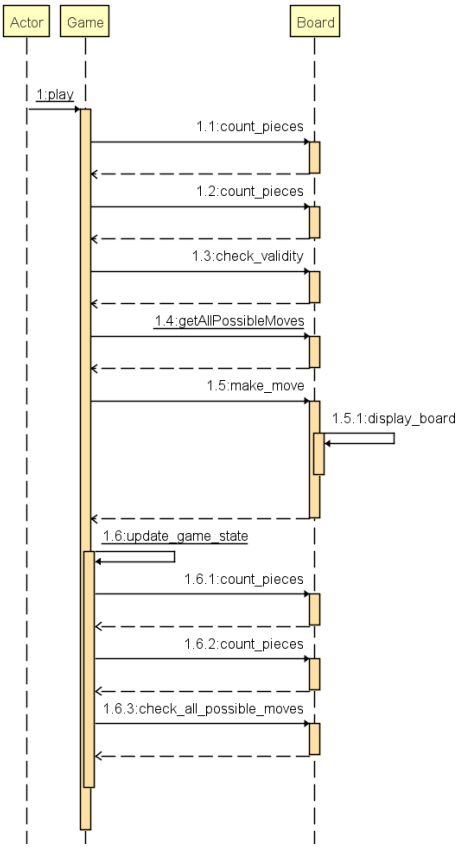
Exercise 2 - A Checkers Game – Unit Tests: IN! (30 pts)

1. **Why our classes are important, Responsibilities**
    a. **Game** Class: Game Class is important because it acts as the entry point for the player. It's method update_game_state is responsible for checking if the game should still be running or if it is over. It's play class holds the code used to progress through the game/rounds by making calls to other classes and instantiating other objects.



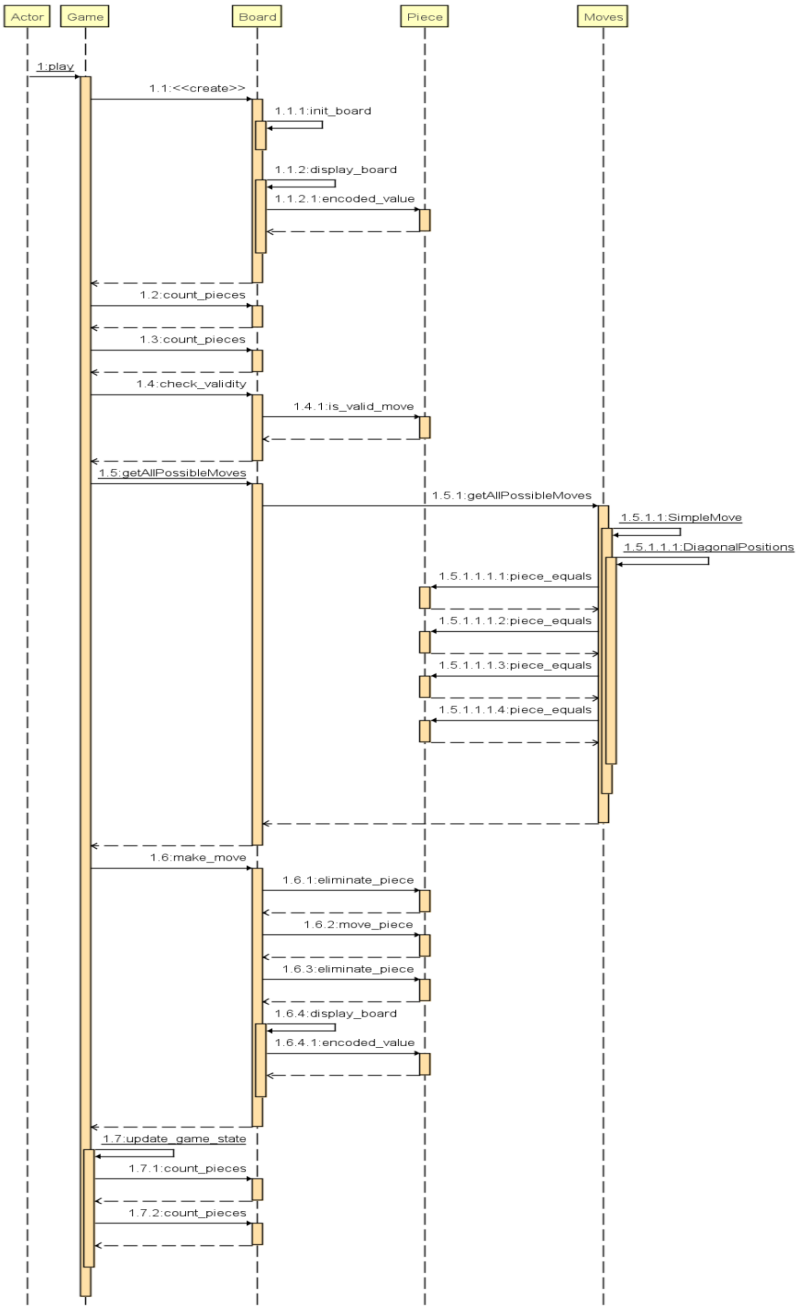
This is also apparent from the above sequence diagram: Starting with Game.play, every other class is called. Game acts as the coordinator which ultimately brings together & uses the functionality of all other classes to create a usable game that can played by a user.

b. **Board** Class: Board class is responsible for providing a board to store the players pieces and also keeping track of their positions. It provides methods to move pieces on the board and also to return the amount of pieces on the board, so as to not make the board of pieces itself available publicly. It's important because it stores the current state of the game in its board.



As you can see in the sequence diagram, Game heavily depends on / uses Board. Game also calls utils a few times, but those were left out of the graph in order to keep it shorter.

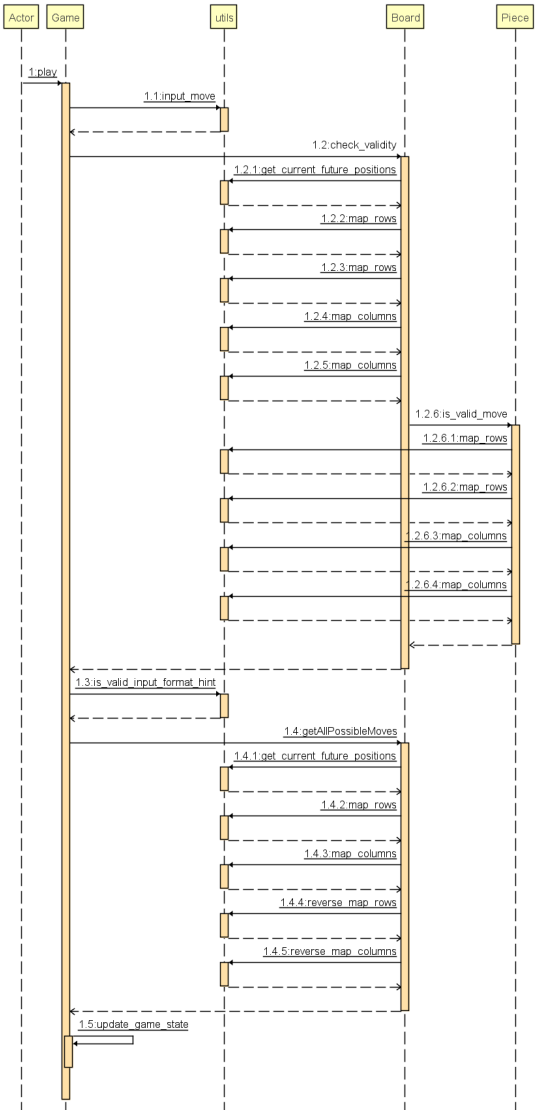
c. **Piece** Class: Pieces are what is stored in the Board. They're important because they provide functionality to the Board. This functionality includes the ability to move a piece without needing to worry about the concrete implementation of moving a piece in the board or if it needs to be promoted to king. It also provides a method to check if a move is valid, which is an important task.



As we can see from the sequence diagram, Piece is needed both to print the board and to count the pieces on the board of each color. Thus Piece is needed to convey the current state of the game to the user, which is vital to the game.

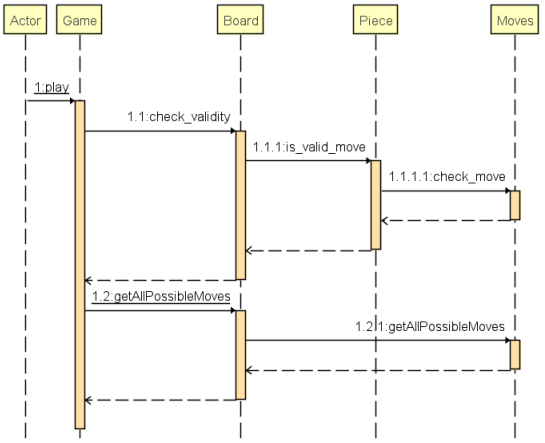
d. **utils** Class: utils consists of static methods used all throughout the code which are stored in a separate class to allow for easier readability of the rest of the codebase. It's responsible for taking input and mapping input - which describes columns and rows using letters and digits- to a format usable to access the board array. It also provides a method to check if input is in a valid format for getting a hint. It is

important because it allows for easier readability of the rest of the code and it also groups methods with a very similar purpose together.



As is apparent from this sequence diagram, utils is heavily used by several Board methods (not all included) and Piece.is_valid_move

e. **Moves** Class: The Moves class has two main responsibilities: 1: Checking if a certain move from one given position to another given position is a legal move. 2: Returning all possible moves for a given Piece. This class is very important because without it, we would have no way of checking if a move input by the user is actually valid.



As we can see from the sequence diagram, Moves is used by both Piece and Board

2. **Unit Tests**

GameTest: Creating tests for the game class was tricky. Game has three methods: main(), which simply calls play, play() and update_game_state(). The only public method is main. Upon calling main, the program enters play() and therefore gets into a loop which runs as long as the game is not over. In order for a test to be able to test anything and the test to pass, this loop would have to end, meaning we would need to play an entire game in the test. Even then, the only thing we could really test is whether, after a sequence of moves which should result in gameover, the program actually prints “Game is Over! Red/White is winner!".

The other big problem was that when entering the loop in play, the game asks the user to input a move. But as tests are automated, we needed to automate this process of inputting moves too. This took us a decent amount of time to get to work, but it works just fine now. Ultimately, we used System.setIn(“input”) before actually calling Game.main. “Input” is a string consisting of multiple separate moves separated by a \n.

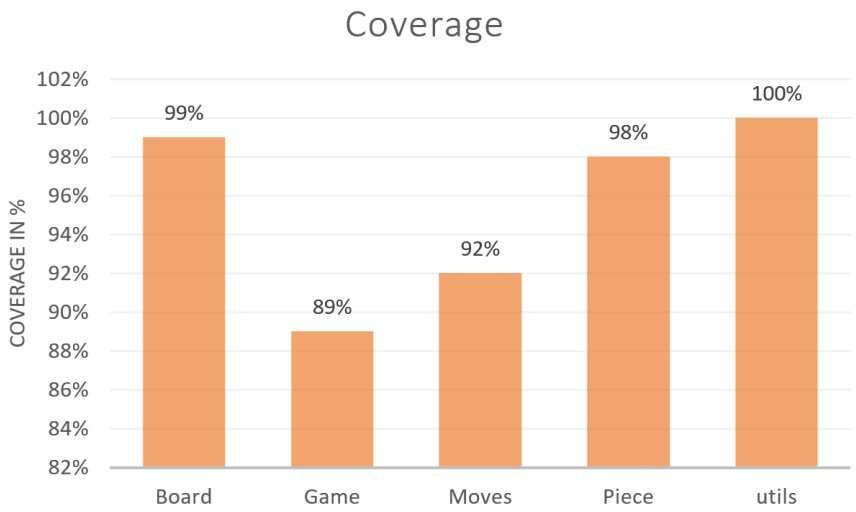
The way that our GameTest works right now is that we have as input a sequence of moves which seek to call as many parts of the GameClass as possible in order to get our line coverage up. This is a primitive approach which doesn’t really test anything, but it achieves the 80% line coverage. If we wanted to actually test something, we would need to encode as input a sequence of moves which leads to a gameover. This would be a very time-consuming task and would still only test such a small fraction of Game, which is why we decided against doing it.
The problem is that if we don’t have an input string that terminates the loop in play, then the test would still run infinitely and the test would never pass/fail. However, Scanner.nextLine throws an error when there are no more \n in the

input, which terminates the program and thus breaks the loop. We use this to our advantage and simply test whether this error gets thrown.

To conclude, we know that this approach is very primitive and not really useful, but we now have our 80% line coverage and a test that finishes.

3. **Testing report**

Line coverage overall: 95% of lines covered



Exercise 3 - A Checkers Game – 20%-Time (30 pts)

4. **Extensions and Improvements**

We had received approval for the following -

1. *We would like to introduce the "Single player" option to our Checkers game. When we start the game, it will first ask the user to choose between "Single player" or "Double player". If the game is Single player, the player can play with a computer. If Double player, two players can play the game.*
2. *We also want to personalize the game a bit. So, once the user chooses the type of Game, the game asks the player or players to enter their names. Once the game knows the names, the player or players will be addressed with their corresponding names during the entire game.*
3. *In addition to these new features, we would like to refactor our code to provide a more improved version of it.*

And a description for our solutions for the above are as follows -

1. The game now starts by asking what “Mode” you want to play in i.e **Single** or **Double** Player modes. And as the mode name suggests, if you enter Single, you play with the Computer and if you enter Double, then you can play with another person.
    a. For Single Player mode, we had to implement a basic AI player. The AI player chooses White pieces and asks you to make the first move. Afterwards it first figures out what pieces can make a move and then randomly selects a piece. Then, it figures out a list of possible moves for the selected piece, and then randomly selects a move. All these are done while making sure we follow the rules of the game.
    b. For Double Player, we refactored the code such that the program works accordingly for Single and Double Player while not having any side effects on the Double Player Mode.

    While implementing the AI, we had to make 2 new classes - **Player** Class and **AI** Class. To integrate the classes, we had to make a lot of changes to the **Game** class and some changes in **Moves** class. While writing the new classes and making the changes, we started with public variables and methods just to be able to get off the ground in terms of progress. Afterwards, we started plugging the gaps and inconsistencies by making sure that the variables and methods that are not needed outside are made private or by implementing getters and setters for variables that are important for other classes to access for their proper functioning.

2. To make the game look more personalized, instead of referring to the players as *Player 1* and *Player 2*, we ask the players to enter their names. Afterwards, the players are referred to with their **name**s throughout the game. In Single Player mode, the computer player is referred to as **Computer** and the human player is referred to by the name they entered.

3. To improve our code and add more features, we refactored the Game class because this is the class that is responsible for the inputs and initiation of the game. In addition to that, we also improved encapsulation by making all the class variables private including some public variables in Player and AI class. We have decided to keep getters and setters in the Player class because we believe it is important to how we have designed the game.

NEW FEATURE
(This is something new that we decided to add later on, so we did not take formal approval for this feature, but we added it anyways)

4. We finally added **Scoreboard**. Every time, player or players make the move, they will be able to see the scoreboard. Scoreboard starts with [0,0] scores for both players in both game modes and scores will increase by 1 every time a player knocks out another player's piece. In this way, the game feels more competitive since players can see their scores.

5. Responsibility Driven Design and UML

Following are the CRC cards of our new system. We only included CRC cards of important classes.

Game	
• Store players information • State design pattern • Piece count • Check input validity	• Player • Context • StartState • StopState • Board • AI

Board	
• Save pieces in 2D array • Get all possible moves and knockout positions	• Pieces • Moves

Piece	
• Moves Pieces • Promotes Pieces • Registered Observer for Observer Pattern • Check move validity • Get Piece's details	• Moves

Player	
• Player Name • Player Color • Player Pieces	• Board

AI	
• Computer Player in Single Mode	• Game • Board • Moves

Important New Classes:

Player

It is reasonable to assume that the Player class will hold information about the player name, and color and would provide either attributes or setters and getters for the same. It also provides information on what pieces for a given player still exist on the board.

When it comes to collaboration, Player class collaborates with Board class. It needs the information about the existing state of the board that holds information on all the pieces that exist on the board for both players. Using the board attribute it is able to distinguish between pieces of the players or the player and the AI based on the color information provided.

AI

AI class is responsible for automatically making moves when the player chooses to play in Single Player mode.

It collaborates with Game, Board and Moves class. Game class needs the AI object to act as the other player in Single Player mode. It is responsible for parsing the input move from the AI and passing it on to validity checks and finally the movement of a piece.

Using Board Class, the AI class knows the pieces of the opponent and essentially what moves are possible.

Using Moves class, the AI actually makes the pieces move.

Note: Class Diagram & Sequence Diagram can be referred to in the solution for Exercise 1.
