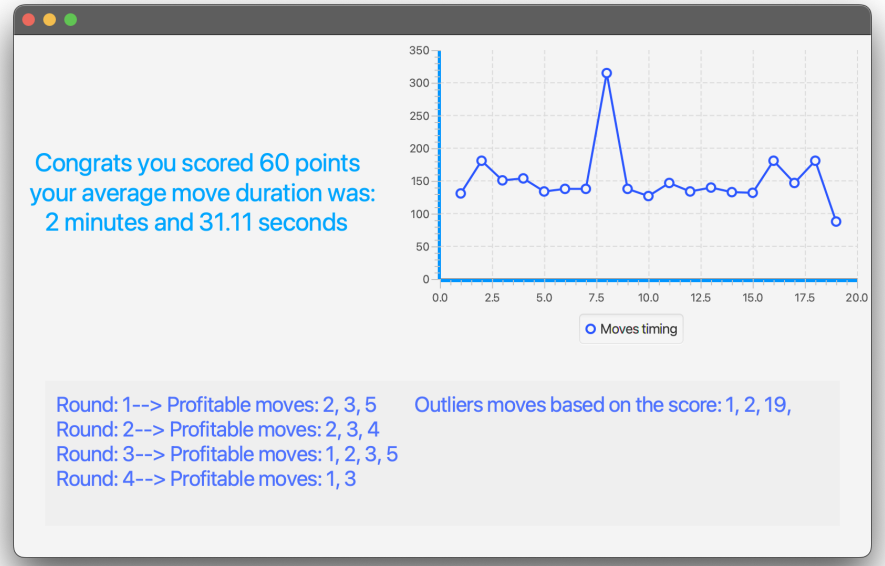
# Documentation Statistics Screen

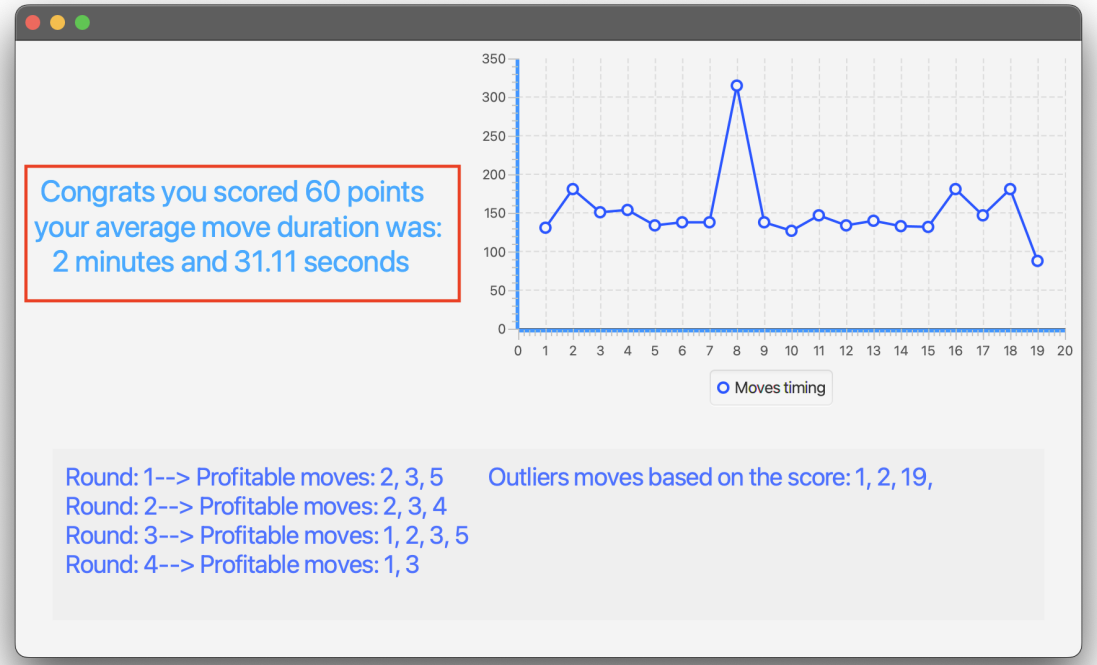
**Disclaimer**: This deliverable has been made with model data. The game logic isn't ready to collect data yet.




Presented here is the current version of our game's statistics screen, which is designed to display at the end of each match. While this is not the final version of the screen, it has been optimised for functionality rather than aesthetic appeal. However, we intend to refine its layout before the conclusion of sprint number 3.

This statistics screen provides players with an overview of their in-game performance, including the points earned during the match, the average duration of a move, a chart that displays the duration of each move in seconds, the most profitable moves of each round, and outlier moves determined by their score.

Subsequently, this document will delve deeper into each component of the mentioned screen and explore the corresponding Java code used to implement them. Our analysis will concentrate exclusively on the "Statistics" class, as this is where the underlying logic of the screen is generated.




For this part of the screen, we used the `getAverageMoveDuration()` and `getFinalScore()` methods.

```
public String getAverageMoveDuration(){
    String average_move_duration = null;
    try {
        ResultSet resultSet = statement.executeQuery("select extract('minutes' from avg(end_time - start_time)) || ' minutes and ' ||\n" +
                "       to_char(extract('seconds' from avg(end_time - start_time)), '99.99') || ' seconds'\n" +
                "from move\n" +
                "where game_id = '1';");
        while(resultSet.next()){
            average_move_duration = resultSet.getString(1);
        }

    } catch (Exception e) {
        System.out.println("Error executing the more_avg method");
        e.printStackTrace();
    }

    return average_move_duration;
}
```

The `getAverageMoveDuration()` returns a string value. The method uses a SQL query to calculate the average duration of moves in a game.

The SQL query calculates the difference between the "end_time" and "start_time" of each move, then calculates the average duration in minutes and seconds. The result is a string in the format of "X minutes and Y seconds".

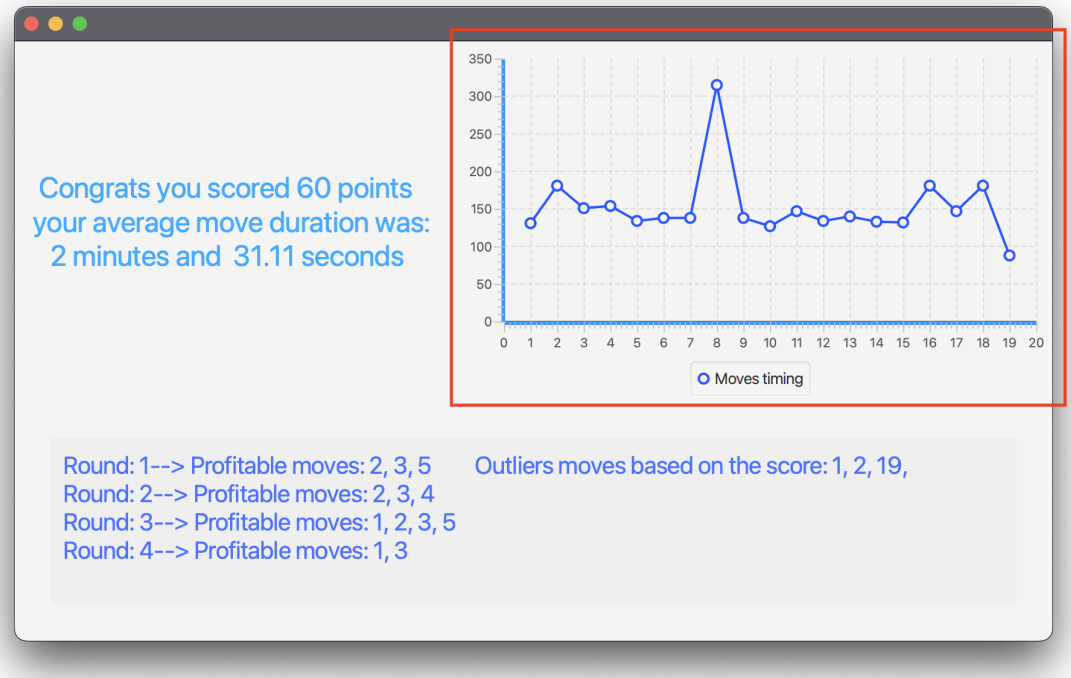
The method executes the query using a statement object and saves the result in a ResultSet object. It then retrieves the result from the ResultSet object and assigns it to the "average_move_duration" variable, which is later returned as a String.

```java
public String getFinalScore(){
        String final_score = null;
        try {
            ResultSet resultSet = statement.executeQuery("select points\n" +
                    "from move\n" +
                    "where game_id = '1'\n" +
                    "order by move_id desc fetch first row only;");

            while(resultSet.next()){
                final_score = resultSet.getString(1);
            }
        } catch (SQLException e){
            e.printStackTrace();
        }
        return final_score;
    }
```

The `getFinalScore()` method uses a SQL query to retrieve the last recorded “points” from a game, which represent the final score of a player

.

The method executes the query using a statement object and saves the result in a ResultSet object. It then retrieves the result from the ResultSet object and assigns it to the "final_score" variable. If an error occurs, the method prints the stack trace. Finally, it returns the "final_score" string value which is used by the View of the Application.



For this part of the screen, we used a Java FX node called LineChart. The method responsible for getting the data for the chart is `getMoveChartValues()`.

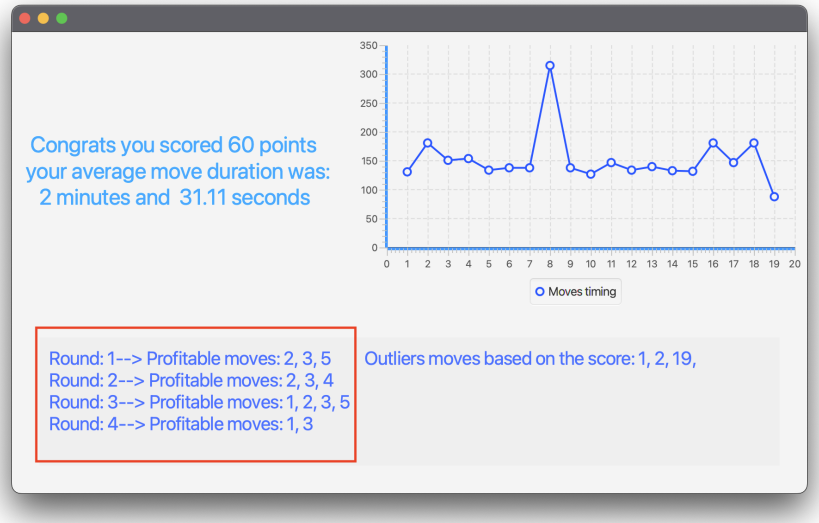
```java
public Map<Integer, Integer> getMoveChartValues(){
    Map<Integer, Integer> move_dur_map = new HashMap<Integer,Integer>();
    try{
        ResultSet resultSet = statement.executeQuery("select extract('minutes' from end_time-start_time)*60 + extract('seconds' from end_time-start_time)\n" +
                "from move\n" +
                "where game_id  = '1'\n" +
                "order by move_id;");
        int i = 1;
        while(resultSet.next()){
            move_dur_map.put(i++, resultSet.getInt(1));
            }
        } catch (Exception e) {
        System.out.println("Error executing the move_dur_round");
        e.printStackTrace();
        }

    return move_dur_map;
    }
```

This method returns a map of integer values. The map contains the duration of each move in a game, indexed by move number.

Inside the method, the code executes a SQL query using a statement object. The query selects the duration of each move in the game with the ID of '1'. The durations are calculated by subtracting the start time from the end time of each move, and converting the result to seconds.

The results of the query are stored in a ResultSet object, which is then used to populate the move_dur_map map. The i variable is used as a counter to keep track of the move number, and the resultSet.getInt(1) method call retrieves the duration of the move from the query results.

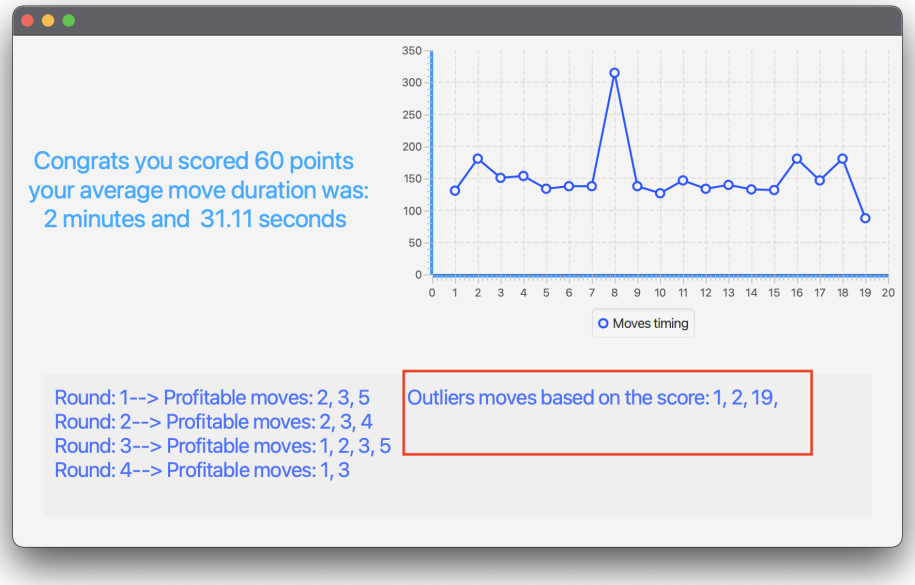
Finally, the method returns the move_dur_map map with the duration of each move in the game. The map is later passed to the LineChart which displays it on the screen.



For this part we used the `getMostProfitableMoves()` method. Later we will display those informations in a table, but for now it displayed as a text.

```java
public String getMostProfitableMoves() {
    Map<Integer, String> prof_moves_map = new HashMap<Integer, String>();
    try {
        ResultSet resultSet = statement.executeQuery("SELECT round_id, move_number\n" +
                "FROM (SELECT round_id, points, move_number, ROW_NUMBER() OVER (ORDER BY round_id,move_number) AS row_num\n" +
                "      FROM move\n" +
                "      where game_id = '1') t1\n" +
                "WHERE points = (SELECT points\n" +
                "                FROM (SELECT points, ROW_NUMBER() OVER (ORDER BY round_id,t1.move_number) AS row_num\n" +
                "                      FROM move) t2\n" +
                "                WHERE t1.row_num = t2.row_num + 1)");

        while (resultSet.next()) {
            if (prof_moves_map.containsKey(resultSet.getInt(1))) {
                String str = prof_moves_map.get(resultSet.getInt(1)) + ", " + resultSet.getString(2);
                prof_moves_map.replace(resultSet.getInt(1), str);
                } else {
                prof_moves_map.put(resultSet.getInt(1), resultSet.getString(2));
                }
            }
        } catch (Exception e) {
        System.out.println("Error executing the most_prof_move");
        e.printStackTrace();
        }

    StringBuilder str = new StringBuilder();
    for (Map.Entry<Integer, String> entry : prof_moves_map.entrySet()) {
        str.append("Round: ").append(entry.getKey()).append("--> Profitable moves: ").append(entry.getValue()).append("\n");
        }

    return str.toString();
    }
```

This method retrieves necessary data from the database using a SQL query. In the game, a move is considered profitable if the player doesn't earn any points. The SQL query compares the player's score after each move with the previous move. If the player doesn't earn any points, the move is considered profitable and is stored in the result set. The method returns a map where the round number is the key, and the string of profitable moves from that round is the value of the map.



For this part of the screen we used the `getOutliersRounds()` method, along with a method from a different class `getOutliers(List<Double> data)`

```java
public String getOutliersRounds(){
    StringBuilder outliers_moves;
    List<Double> input = new ArrayList<Double>();
    List<Integer> round_of_move = new ArrayList<Integer>();
    try{
        ResultSet resultSet = statement.executeQuery("select move_id, points\n" +
                "from move\n" +
                "where game_id = '1'\n" +
                "order by 1;");

        while (resultSet.next()){
            round_of_move.add(resultSet.getInt(1));
            input.add(resultSet.getDouble(2));
            }
        connection.close();
        } catch (SQLException e){
        System.out.println("Error retrieving outliers");
        e.printStackTrace();
        }

    outliers_moves = new StringBuilder();
    List<Double> outliers = OutlierDetector.getOutliers(input);
    for (Double outlier : outliers){
        outliers_moves.append(round_of_move.get(input.indexOf(outlier))).append(", ");
        }

    return String.valueOf(outliers_moves);

    }
```

This method executes a SQL query on a database to retrieve the move_id and points for all moves, then adds the move IDs and points to the round_of_move and input lists, respectively. It uses the `getOutliers(List<Double> data)` method to obtain a List of outlier values from the input list.Then it iterates through the outliers list, and for each outlier value, it retrieves the corresponding round number from the round_of_move list by using the indexOf method to find the index of the outlier value in the input list.

```java
public class OutlierDetector {

    public static List<Double> getOutliers(List<Double> data) {
        List<Double> outliers = new ArrayList<>();
        Collections.sort(data);

        double median;
        int size = data.size();
        if (size % 2 == 0) {
            median = (data.get(size / 2 - 1) + data.get(size / 2)) / 2.0;
        } else {
            median = data.get(size / 2);
        }

        double q1;
        int q1Index = size / 4;
        if (size % 4 == 0) {
            q1 = (data.get(q1Index - 1) + data.get(q1Index)) / 2.0;
        } else {
            q1 = data.get(q1Index);
        }

        double q3;
        int q3Index = size * 3 / 4;
        if (size % 4 == 0) {
            q3 = (data.get(q3Index - 1) + data.get(q3Index)) / 2.0;
        } else {
            q3 = data.get(q3Index);
        }

        double iqr = q3 - q1;

        for (Double d : data) {
            if (d < median - 1.5 * iqr || d > median + 1.5 * iqr) {
                outliers.add(d);
            }
        }
        List<Double> output = outliers;
        return output;
    }

}
```

This method calculates the median value of the “data” list by checking its size and whether it's even or odd.It calculates the first quartile value (q1) of the data list by finding the value at the 25th percentile, calculates the third quartile value (q3) of the data list by finding the value at the 75th percentile.After this it calculates the interquartile range (iqr) by subtracting q1 from q3, iterates through each value in the data list, and for any value that is less than median - 1.5 * iqr or greater than median + 1.5 * iqr, and adds that value to the outliers

list.It assigns outliers to a new list called “output” and returns it, which contains any detected outlier values in the input data list.