Final DAI document
Finalizing game statistics and Rule based intelligence

Rubric Line	Score	Improvements
------Game statistics----------	----------	----------
Creating the tables and ERD (DDL)	Nearly meets expectations	The creation of the tables is now explained in the documentation and is also used at the start of every game (if something happens with the tables on the database)
Game logic and handling Data (DML)	Meets expectations	The code for saving informations from the game is now present in the documentation
Processing game data	Meets expectations	No improvements
Visual data representation	Meets expectations	
------Rule Base Intelligence----	----------	----------
Rule base:	INVALID	The AI was not ready for the first submission
MVP:	INVALID	The AI was not ready for the first submission
This criterion is linked to a learning outcome Expert rules:	INVALID	The AI was not ready for the first submission

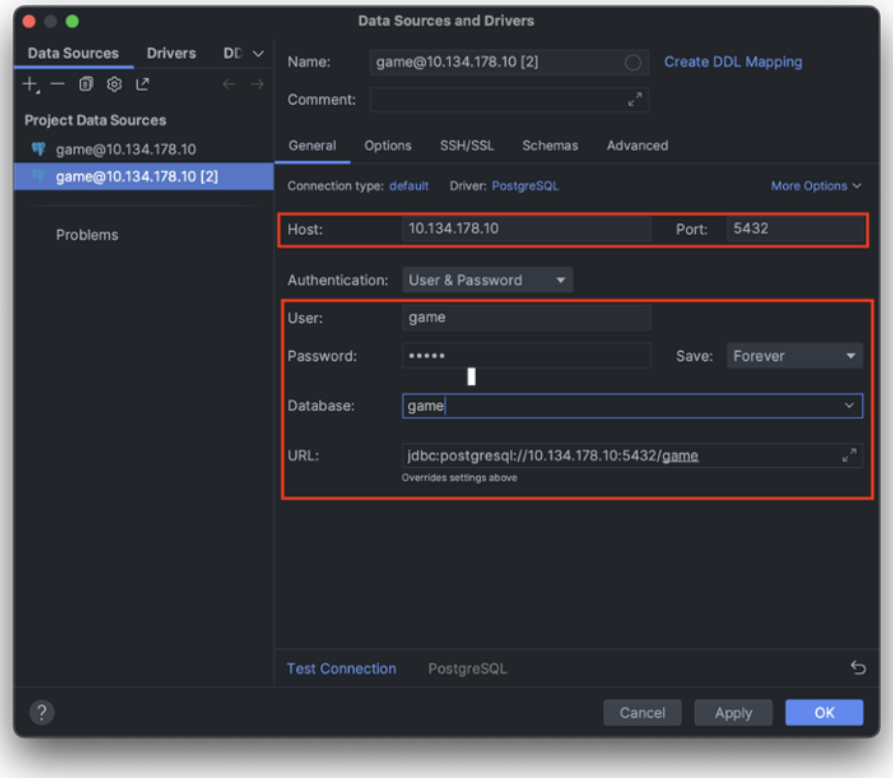
Game statistics

Documentation Statistics Screen Team 10

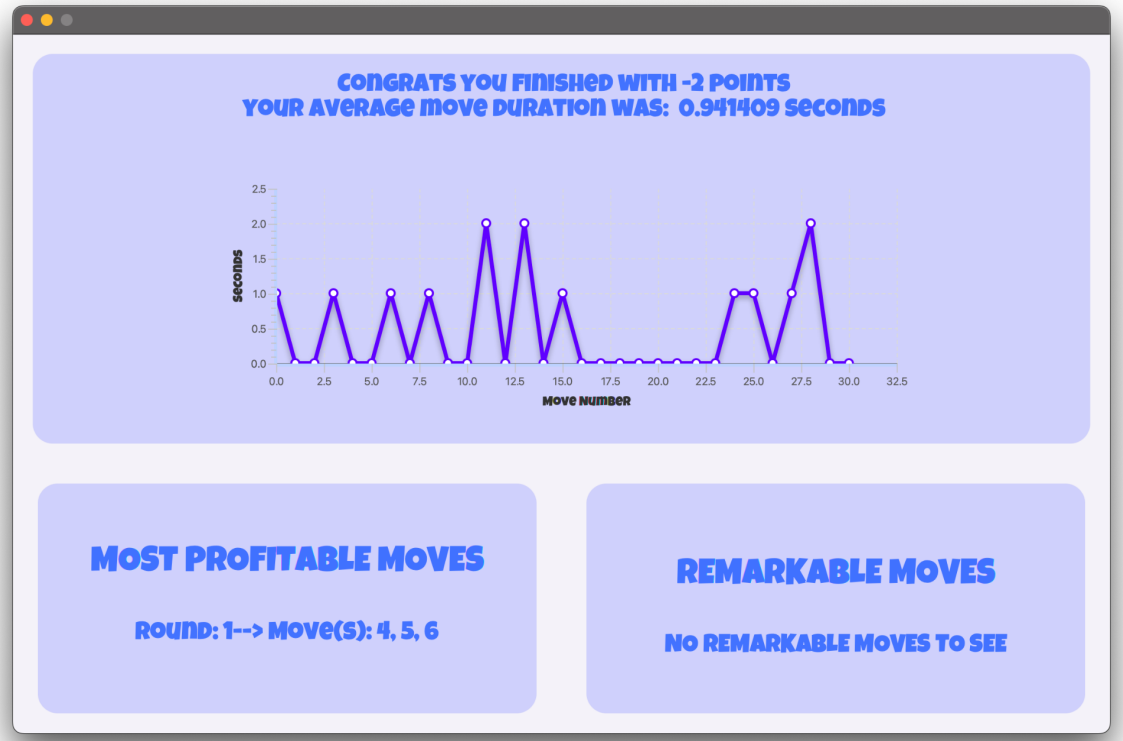
Game: Take5!
User: game
Password: 7sur7
Database name: game
Server: 10.134.178.10
Url for java jdbc: jdbc:postgresql://10.134.178.10:5432/

Important ! In order to connect to the database you must be connected to the Kdg intranet .

To connect to the database, set-up a new “PostgreSQL Data Source” in Intellij with the information provided above.



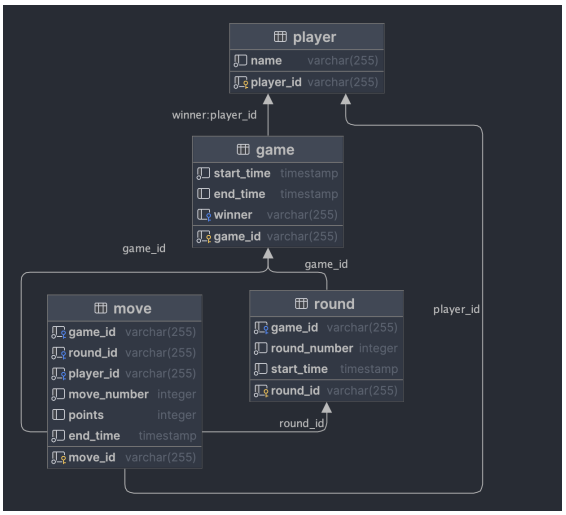
Documentation Statistics



Presented here is the current version of our game's statistics screen, which is designed to display at the end of each match. 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.

ERD



The ERD consists of multiple tables that collectively represent the structure and relationships of a game-related database.These tables collectively capture and store various aspects of game-related data, including game details, player information, round information, and move details. The relationships between these tables enable the establishment of meaningful associations and dependencies, facilitating the management and analysis of game-related information in a structured manner.

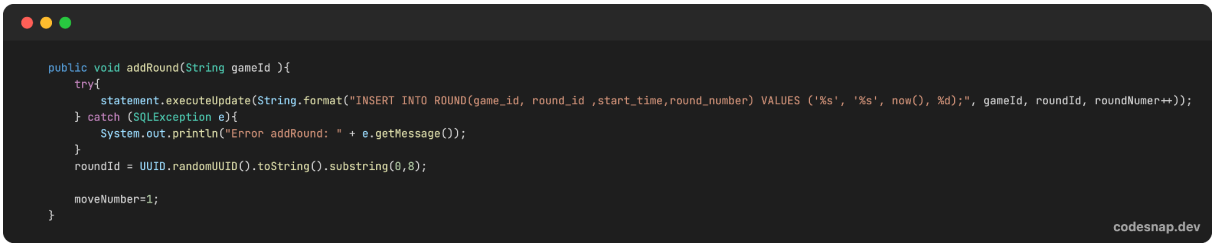
The tables are created (if they don't exist) at the start of every game using the `createTables()` method from the DatabaseManager class.

Create Tables

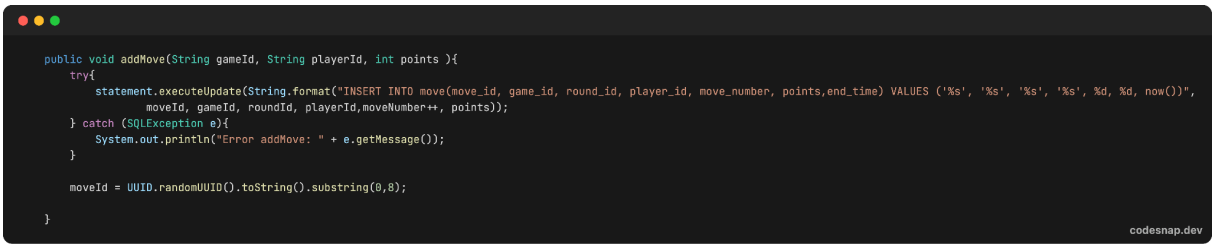
```
public void createTables(){
        try{
            statement.executeUpdate("CREATE TABLE IF NOT EXISTS player\n" +
                    "(\n" +
                    "    player_id VARCHAR(255) PRIMARY KEY,\n" +
                    "    name      VARCHAR(255) NOT NULL\n" +
                    ");\n" +
                    "\n" +
                    "CREATE TABLE IF NOT EXISTS game\n" +
                    "(\n" +
                    "    game_id    VARCHAR(255) PRIMARY KEY,\n" +
                    "    start_time TIMESTAMP NOT NULL,\n" +
                    "    end_time   TIMESTAMP,\n" +
                    "    winner     VARCHAR(255)\n" +
                    ");\n" +
                    "\n" +
                    "ALTER TABLE game\n" +
                    "    ADD CONSTRAINT fk_player FOREIGN KEY (winner) REFERENCES player (player_id) ON DELETE CASCADE;\n" +
                    "\n" +
                    "CREATE TABLE IF NOT EXISTS round\n" +
                    "(\n" +
                    "    round_id     VARCHAR(255) PRIMARY KEY,\n" +
                    "    game_id      VARCHAR(255) NOT NULL,\n" +
                    "    round_number INTEGER      NOT NULL,\n" +
                    "    start_time   TIMESTAMP    NOT NULL\n" +
                    ");\n" +
                    "\n" +
                    "CREATE TABLE IF NOT EXISTS move\n" +
                    "(\n" +
                    "    move_id     VARCHAR(255) PRIMARY KEY,\n" +
                    "    game_id     VARCHAR(255) NOT NULL,\n" +
                    "    round_id    VARCHAR(255) NOT NULL,\n" +
                    "    player_id   VARCHAR(255) NOT NULL,\n" +
                    "    move_number INTEGER      NOT NULL,\n" +
                    "    points      INTEGER,\n" +
                    "    end_time    TIMESTAMP    NOT NULL\n" +
                    ");\n");
        } catch (Exception e){
            System.out.println("Error when creating tables" + e.getMessage());
        }
    }
```

codesnap.dev

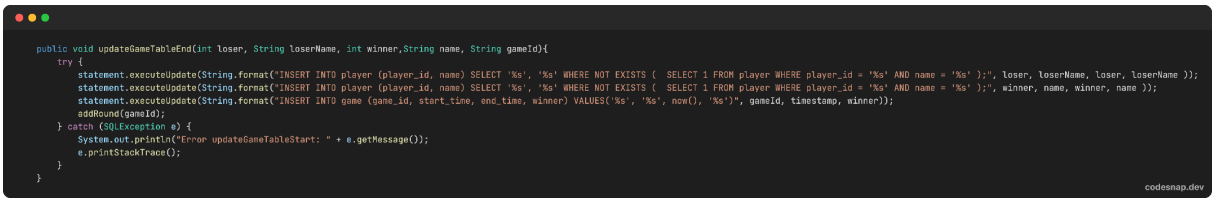
It creates the player table with two columns: player_id (a primary key) and name.Next, it creates the game table with four columns: game_id (a primary key), start_time, end_time, and winner. The winner column references the player_id column in the player table as a foreign key with the ON DELETE CASCADE option, meaning that if a player is deleted, the corresponding games where they are the winner will also be deleted.The round table is created with four columns: round_id (a primary key), game_id, round_number, and start_time. The game_id column references the game_id column in the game table as a foreign key with the ON DELETE CASCADE option.Lastly, the move table is created with seven columns: move_id (a primary key), game_id, round_id, player_id, move_number, points, and end_time. The game_id, round_id, and player_id columns reference the corresponding primary keys in their respective tables as foreign keys with the ON DELETE CASCADE option.



At the start of every round, the `addRound()` method is called. This method saves the game id of the game, a unique round id for the round, the current time when the round starts, and a roundNumber. The roundNumber is incremented each time a round is added. If there is an issue with saving the information, it prints an error message.Next, it generates a random roundId using a unique identifier called UUID. The round id is a series of characters that uniquely identifies the round. In this code, it takes the first 8 characters of the generated ID.Finally, it sets the moveNumber variable to 1, indicating that it's the first move of the newly added round.



After every move the player has made, the addMove() method is called. It executes an SQL query to insert a new row into a table called move. The values being inserted are moveId, gameId, roundId, playerId, moveNumber, points, and the current timestamp (now()). The moveNumber is incremented after each insertion.It generates a random moveId using UUID.randomUUID().toString(), and then takes a substring of the first 8 characters from the generated ID. This ensures that the moveId is a unique identifier for the move.



```
public void updateGameTableEnd(int loser, String loserName, int winner,String name, String gameId){
    try {
        statement.executeUpdate(String.format("INSERT INTO player (player_id, name) SELECT '%s', '%s' WHERE NOT EXISTS (  SELECT 1 FROM player WHERE player_id = '%s' AND name = '%s' );", loser, loserName, loser, loserName ));
        statement.executeUpdate(String.format("INSERT INTO player (player_id, name) SELECT '%s', '%s' WHERE NOT EXISTS (  SELECT 1 FROM player WHERE player_id = '%s' AND name = '%s' );", winner, name, winner, name ));
        statement.executeUpdate(String.format("INSERT INTO game (game_id, start_time, end_time, winner) VALUES('%s', '%s', now(), '%s')", gameId, timestamp, winner));
        addRound(gameId);
    } catch (SQLException e) {
        System.out.println("Error updateGameTableStart: " + e.getMessage());
        e.printStackTrace();
    }
}
```

codesnap.dev

At the end of every game the updateGameTableEnd() method is called. This method saves the information for both the winner and the loser of the game (one of them being the ai) if they don’t already exist in the table. After this the table with the information of the current game is updated. The gameID the start and the end of the game is saved in the table, along with the id of the winner.

Explanation Parts of The Stats Screen

For this part of the screen, we used the `getAverageMoveDuration()` method.

```
public String getAverageMoveDuration(String gameId){
    String average_move_duration = null;
    try {
        ResultSet resultSet = statement.executeQuery(String.format("WITH move_times AS (\n" +
                "    SELECT end_time - LAG(end_time) OVER (ORDER BY end_time) AS move_time\n" +
                "    FROM move\n" +
                "    WHERE game_id = '%s'\n" +
                ")\n" +
                "SELECT extract('seconds' from AVG(move_time)) || ' seconds' AS avg_move_time\n" +
                "FROM move_times\n" +
                "WHERE move_time IS NOT NULL;", gameId));
        while(resultSet.next()){
            average_move_duration = resultSet.getString(1);
        }

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

    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.

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(String gameId){
    Map<Integer, Integer> move_dur_map = new HashMap<>();
    try{
        ResultSet resultSet = statement.executeQuery(String.format("WITH move_times AS (\n" +
                "    SELECT end_time - LAG(end_time) OVER (ORDER BY end_time) AS move_time\n" +
                "    FROM move\n" +
                "    WHERE game_id = '%s'\n" +
                "    ORDER BY move_id\n" +
                ")\n" +
                "SELECT extract('minutes' from move_time)*60 + extract('seconds' from move_time)\n" +
                "FROM move_times\n" +
                "WHERE move_times IS NOT NULL;", gameId));
        int i = 0;
        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;
}
```

codesnap.dev

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<>();
    try {
        ResultSet resultSet = statement.executeQuery(String.format("""
                SELECT round.round_number, move_number
                FROM (SELECT round_id, points, move_number, ROW_NUMBER() OVER (ORDER BY round_id,move_number) AS row_num
                      FROM move
                      where game_id = '%s') t1
                join round on round.round_id = t1.round_id
                WHERE points = (SELECT points
                                FROM (SELECT points, ROW_NUMBER() OVER (ORDER BY round_id,t1.move_number) AS row_num
                                      FROM move) t2
                                WHERE t1.row_num = t2.row_num +1);""", gameId));

        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("--> Move(s): ").append(entry.getValue()).append("\n");
    }

    return str.toString();
}
```

codesnap.dev

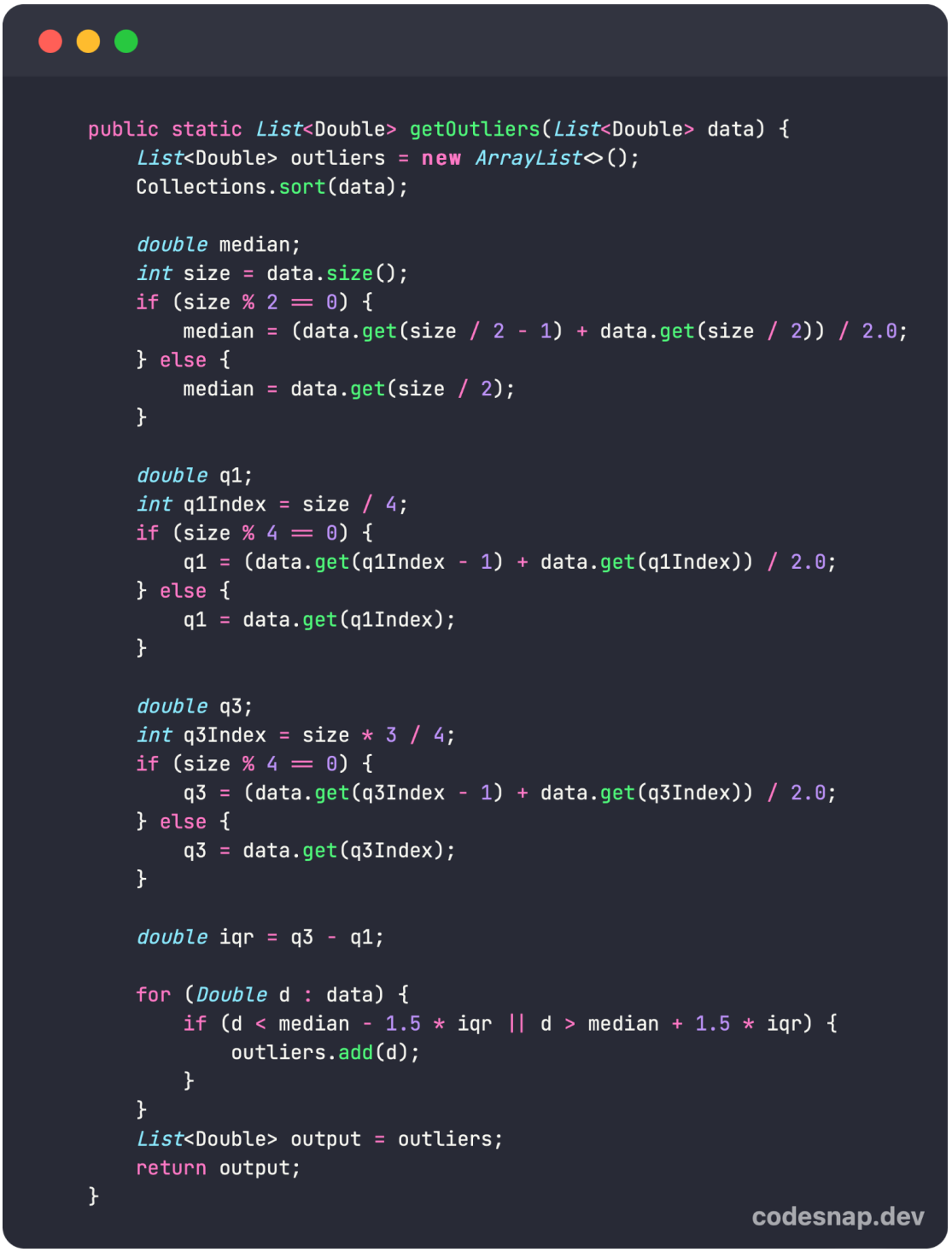
This method retrieves necessary data from the database using a SQL query. In the game, a move is considered profitable if the player doesn't earn any points. The SQL query compares the player's score after each move with the previous move. If the player doesn't earn any points, the move is considered 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)`

```
public String getOutliersRounds(){
    StringBuilder outliers_moves;
    List<Double> input = new ArrayList<>();
    List<String> id_of_move = new ArrayList<>();
    try{
        ResultSet resultSet = statement.executeQuery(String.format("""
                select move_id, points
                from move
                where game_id = '%s'
                order by end_time;""", gameId));

        while (resultSet.next()){
            id_of_move.add(resultSet.getString(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(id_of_move.get(input.indexOf(outlier))).append(", ");
    }

    return String.valueOf(outliers_moves);

}
```

codesnap.dev

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.



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.

Rule based intelligence

The Artificial Intelligence behind the computer player of our game is constructed around a simple, however, effective strategy. Furthermore, it uses the template for playing cards of the human player.

There are three classes which are responsible for the AI behaviour and style of playing the game. The most important one being the “AiPlayer” class. This class extends the “Player” class which provides the methods that are needed for both the AI and human player

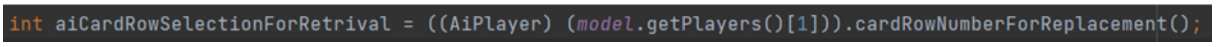
```java
package model;

Undowl +2
public class AiPlayer extends Player {

    2 usages  vladbuinceanu
    public AiPlayer(String name, int counterPoints) { super(name, counterPoints); }

    Undowl
    @Override
    Card chooseCard(int chosen) { return null; }
```

In this class, multiple functions are written so that the AI can behave properly.

```java
    @Override
    public void placeCard(Card card, int row) {

    }

    1 usage  Undowl
    public int cardPlayable() {
        int minCardCol = 105;
        int minColIndex = 0;
        for (int i = 0; i < table.cardRowsSize; i++) {
            Card lastCard = table.cardRows[i].get(table.cardRows[i].size() - 1);
            if (lastCard.getNumber() < minCardCol) {
                minCardCol = lastCard.getNumber();
                minColIndex = i;
            }
        }

        int minCardHand = 105;
        int minCardHandIndex = 0;

        for (int i = 0; i < this.getHand().getCards().size(); i++) {
            if (this.getHand().getCards().get(i).getNumber() < minCardHand &&
                    this.getHand().getCards().get(i).getNumber() > minCardCol) {
                minCardHand = this.getHand().getCards().get(i).getNumber();
                minCardHandIndex = i;
            }
        }
        return minCardHandIndex;
    }
```

The method/function called “cardPlayable” is the most important one amongst the others. It generates an integer that is corresponding to a card in our hand by its position. In other words, we extract each card from the AI players hand, find which of them has the lowest numeration and get its position in the hand. For example, the positions possible are from 1 to 9, therefore, if we receive the integer corresponding to the position it is easy to extract the card object afterwards from the “hand” array list.

The function “cardRowNumberForReplacement” is responsible for helping our AI collect rows of cards. It essentially goes over the board, scanning the four rows of cards and providing the index of the row with the lowest number of cards in it. This information is being processed by the model and provided to the “dmgCalculationAI” function and “getAllCardsFromRow” function.



```
model.dmgCalculationAI(aiCardRowSelectionForRetrival, model.getPlayers()[1]);
model.getAllCardsFromRow(aiCardRowSelectionForRetrival);
```

```
2 usages   Undowl
public void dmgCalculationAI (int rowIndex,Player AiPlayer){
    DmgCalculator dmgCalculator= new DmgCalculator();
    if (rowIndex>3 || rowIndex<0 ){
        rowIndex=0;
    }
    for (int i=0; i<cardRows[rowIndex].size(); i++){
        int bulls= cardRows[rowIndex].get(i).getBulls();
        dmgCalculator.takeDmgAI(bulls,  playingTable: this);
        System.out.println(bulls);
    }
}
```

All of the other functions used are the same as the human players functions.