Mutation testing report

In order to assess and eventually improve the quality of our test suite we also added the *pitest plugin* for mutation testing and we generated the mutation coverage for our packages.

```
gradle-pitest-plugin:1.4.5
```

As some of the packages are GUI related and therefore the code is not tested via Junit tests, the packages we were interested in were *client, objects and utilities*.

For the Client Package we have got the following results:

Pit Test Coverage Report

Package Summary

client

Number of Classes	Line Coverage		Mutation Coverage	
10	97%	260/269	72%	98/137

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
GameDetails.java	100%	38/38	100%	16/16
JdbcSingleton.java	0%	0/9	0%	0/4
Leaderboard.java	100%	32/32	76%	16/21
LeaderboardDaoMySql.java	100%	26/26	64%	7/11
LeaderboardEntry.java	100%	21/21	100%	10/10
User.java	100%	36/36	100%	19/19
UserAuthenticationMySql.java	100%	29/29	67%	10/15
UserDaoMySql.java	100%	10/10	100%	5/5
UserGameTrackerMySql.java	100%	54/54	47%	14/30
UserRegistrationMySql.java	100%	14/14	17%	1/6

The mutation coverage is in general good, meaning that the implemented tests were already efficiently testing the classes.

In the classes with lower mutation testing (UserRegistrationMySql or UserGameTrackerMySql) the lower score is caused by the unkilled mutants in the case of removing the *preparedStatement.set()* instructions as shown in the next picture:

```
36
37 1            preparedStatement.setString(1, username);
38 1            preparedStatement.setString(2, hashedPwd);
39 1            preparedStatement.setString(3, salt);
40 1            preparedStatement.setString(4, nickname);
41
42              preparedStatement.execute();
43
44 1            return true;
45
46          } catch (SQLException e) {
47 1            return false;
48          }
49
50      }
51
52  }
```

Mutations

```
37  1. removed call to java/sql/PreparedStatement::setString → SURVIVED
38  1. removed call to java/sql/PreparedStatement::setString → SURVIVED
39  1. removed call to java/sql/PreparedStatement::setString → SURVIVED
40  1. removed call to java/sql/PreparedStatement::setString → SURVIVED
44  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
47  1. replaced return of integer sized value with (x == 0 ? 1 : 0) → SURVIVED
```

The solution would be to return false in case the preparedStatement is not fully initialised (not all question marks are set) , but we could not find any way to do this.

For the utilities class we have got the following scores:

BcryptHashing.java	100%	5/5	100%	3/3
MathUtils.java	100%	6/6	46%	6/13

After rewriting the test and also introducing new test cases we were able to get 100% mutation coverage for MathUtils class:

MathUtils.java	83%	5/6	100%	13/13

For the classes in the Object Package we have got the following:

Pit Test Coverage Report

Package Summary

objects

Number of Classes	Line Coverage		Mutation Coverage	
7	73%	166/226	46%	91/197

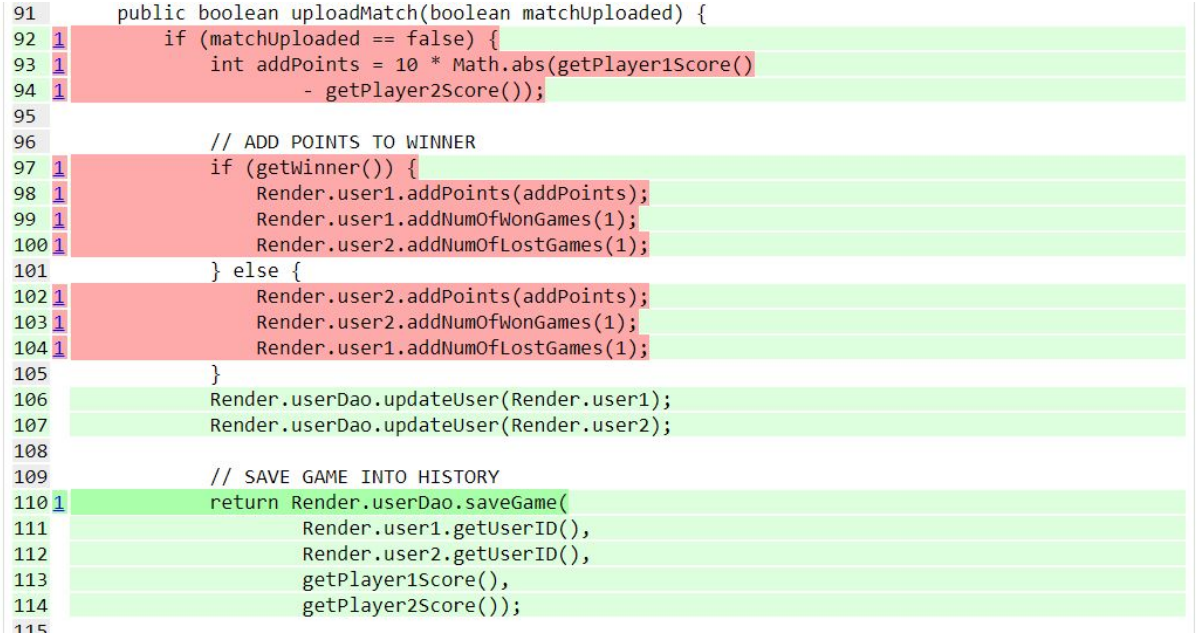
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
GameObject.java	100%	14/14	100%	3/3
GateAlignedState.java	100%	13/13	79%	15/19
Match.java	0%	0/49	0%	0/50
OutOfGatesState.java	100%	10/10	79%	15/19
Puck.java	88%	43/49	64%	21/33
Pusher.java	85%	28/33	53%	19/36
ScoreBoard.java	100%	58/58	49%	18/37

The main reason for a lower score is the presence of drawing methods inside some of these classes that should not be tested with Junit:

```
128                                 float screenWidth, float screenHeight) {
129 1       if (leaderboard == null) {
130             leaderboard = Render.leaderboardDao.getLeaderboard(size);
131         }
132
133 4       informationDrawer.drawText("Player " + winnerNumber() + " Won", (screenWidth / 2) - 150,
134                 screenHeight - 100, 4);
135
136
137
138         int i = 1;
139         for (LeaderboardEntry entry : leaderboard.getLeaderboardList()) {
140
141 1           informationDrawer.drawText(i + ". " + entry.getNickname() + " " + entry.getPoints(),
142                     posX, posY, 4);
143
144 1           posY -= 50;
145 1           i++;
146         }
147
148
149 3       informationDrawer.drawText("Press ENTER to go back to menu", posX - 250, posY - 100, 4);
150     }
151
152     /**
```

There was also the case were some methods were calling other database related methods. In the case that is also depicted below there is nothing that can be done as for example, adding 0 or 10 as the number of points for a player would return the same result from the database method: true if updated in the database and false otherwise. To see if the user details are saved correctly we need to manually check the data in the database.



```
91      public boolean uploadMatch(boolean matchUploaded) {
92  1       if (matchUploaded == false) {
93  1           int addPoints = 10 * Math.abs(getPlayer1Score()
94  1                   - getPlayer2Score());
95
96              // ADD POINTS TO WINNER
97  1           if (getWinner()) {
98  1               Render.user1.addPoints(addPoints);
99  1               Render.user1.addNumOfWonGames(1);
100 1               Render.user2.addNumOfLostGames(1);
101             } else {
102 1               Render.user2.addPoints(addPoints);
103 1               Render.user2.addNumOfWonGames(1);
104 1               Render.user1.addNumOfLostGames(1);
105             }
106             Render.userDao.updateUser(Render.user1);
107             Render.userDao.updateUser(Render.user2);
108
109             // SAVE GAME INTO HISTORY
110 1           return Render.userDao.saveGame(
111                     Render.user1.getUserID(),
112                     Render.user2.getUserID(),
113                     getPlayer1Score(),
114                     getPlayer2Score());
115
```