

TicTacToe Application Report

1 Table of Contents

2	<i>Problem Description</i>	2
3	<i>Implementation Details</i>	2
3.1	The User Interface	2
3.2	Launching the application	4
3.3	The Controller	5
3.4	The Database	6
4	<i>Class Diagram</i>	7
4.1	Class Design	8
5	<i>Unit Tests</i>	8
5.1	Testing the Controller	8
5.2	Testing the database	10

FIGURE 1 - USER INTERFACE	2
FIGURE 2 – DRAW	3
FIGURE 3 – WINNERS	4
FIGURE 4 – DATABASE	6
FIGURE 5 – CLASS DIAGRAM	7
FIGURE 6 – EXTERNAL LIBRARIES	8

2 Problem Description

I must create a two-player game of Tic Tac Toe with a 2D user interface.

The game must be able to allow 2 players to enter their names and play against each other. The game must allow the players to take turns at clicking the buttons in the Tic Tac Toe grid. The game must display “X” when player 1 clicks a button and “O” when player 2 clicks a button. The game must end when one of the players wins. The game must persistently store players scores and must update the score for the winning player.

3 Implementation Details

The TicTacToe application consists of an fxml defined user interface. All events such as mouse clicks are controlled by a Controller class. There is a main class called TicTacToe which launches the application and there is Mysql class which handles all interactions with the Game table in the sqlite database (TicTacToe.mysql). The database allows the game to persist score for the players entered. There are 2 test files. The MysqlTest file tests the methods in the Mysql class. The TicTacToeTest file tests the Controller class. Some methods are tested by creating an instance of the controller class and some are tested using org.testfx.framework.junit.ApplicationTest which is specifically used to test javaFX programs that extend the Application class. This allows the testing of scenarios in the application by automating the clicking of buttons, typing of text, etc.

3.1 The User Interface

The game was built using the javaFX library. The UI is defined using an fxml file called TicTacToe.fxml. This file creates the scaffold for a GridPane for the buttons, text fields and text boxes to be placed into.

Here is an example of the interface before describing the design.

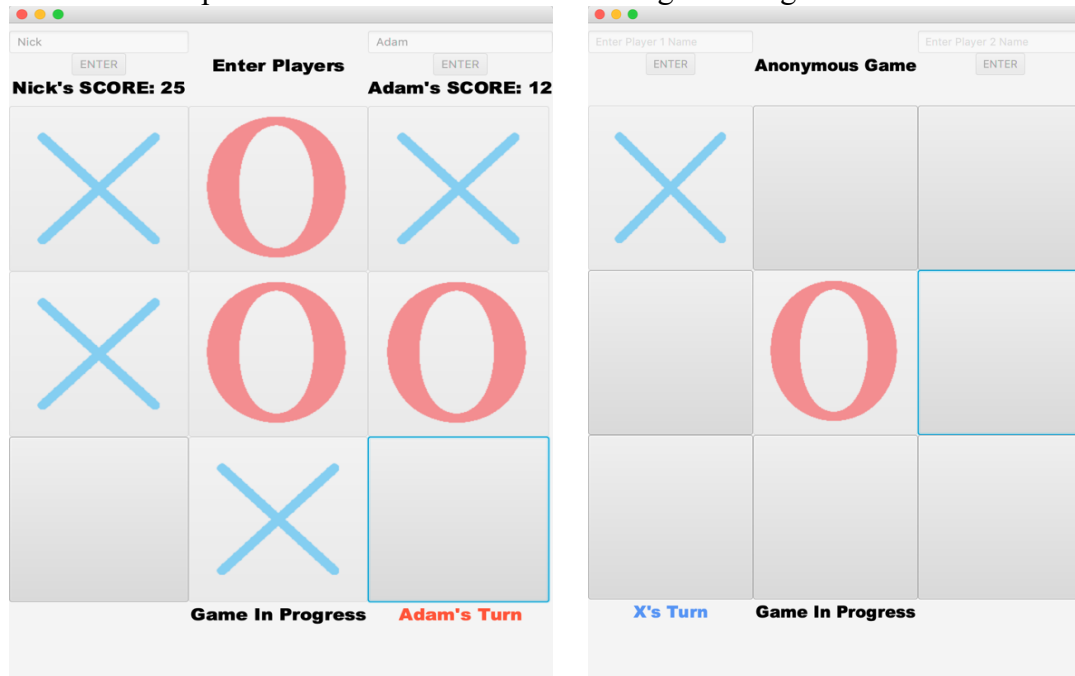


Figure 1 - User Interface

TicTacToe Application Report

The GridPane has five rows.

The first row contains the TextFields for player1 and player2, two enter buttons and a TextBox called instructions. This row allows players to write their names and click enter which will register them to play the game.

Before the game begins, the instruction field will say “Enter Players”.

When players write their names and click ENTER their running total scores will appear under their names. After entering the names and clicking ENTER, those TextFields and buttons become disabled so the user cannot use them again.

If the players want to play an anonymous game without entering their names, then they can just click on any button in the grid. The instruction field will change from “Enter Players” to “Anonymous Game”.

The 2nd to 4th row contains the game grid. These are Buttons that display an “X” image or an “O” image when clicked depending on who’s turn it is. When a button in the game grid it becomes disabled so that button cannot be clicked again.

While the game is in progress the 5th row contains the words “Game In Progress” in the center of the row. If it’s player 1’s turn, then the player’s name will show up on the left side of the row for example “Nick’s Turn”. If it is an anonymous game, it will show “X’s Turn” on the left side. Player 2’s name will show on the right side of the row or will show “O’s Turn” if it’s an anonymous game.

When a game is a draw the player turn information will disappear and the “Game In Progress” will change to “DRAW”. Since all buttons were clicked, they are all disabled, there is no more the user can do. The game is over.

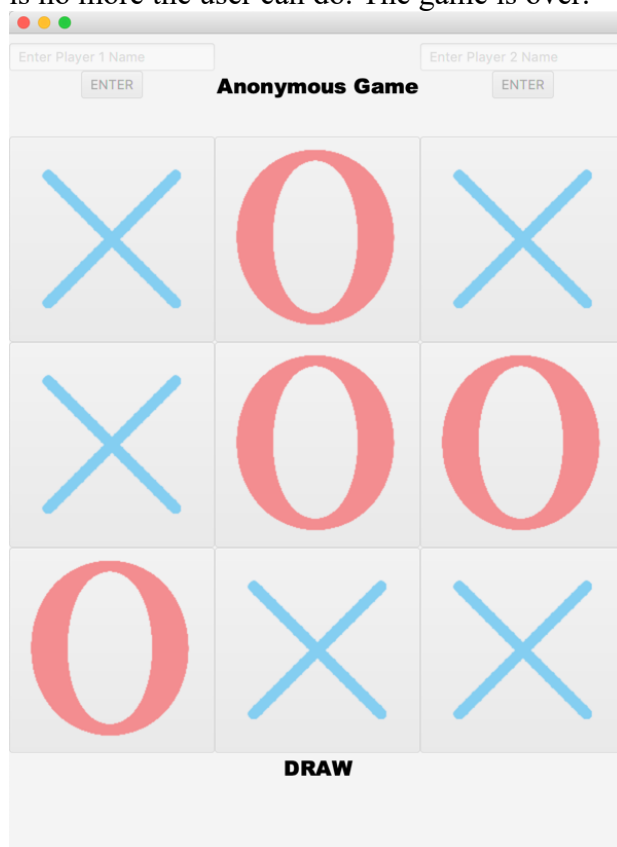


Figure 2 – Draw

When the game is won in an anonymous game, Either “X” or “O” will be the Winner.

Student: Nick Duggan

Student Number: 19201873

TicTacToe Application Report

The text at the bottom changes to show the winner, for example “X WINS”.

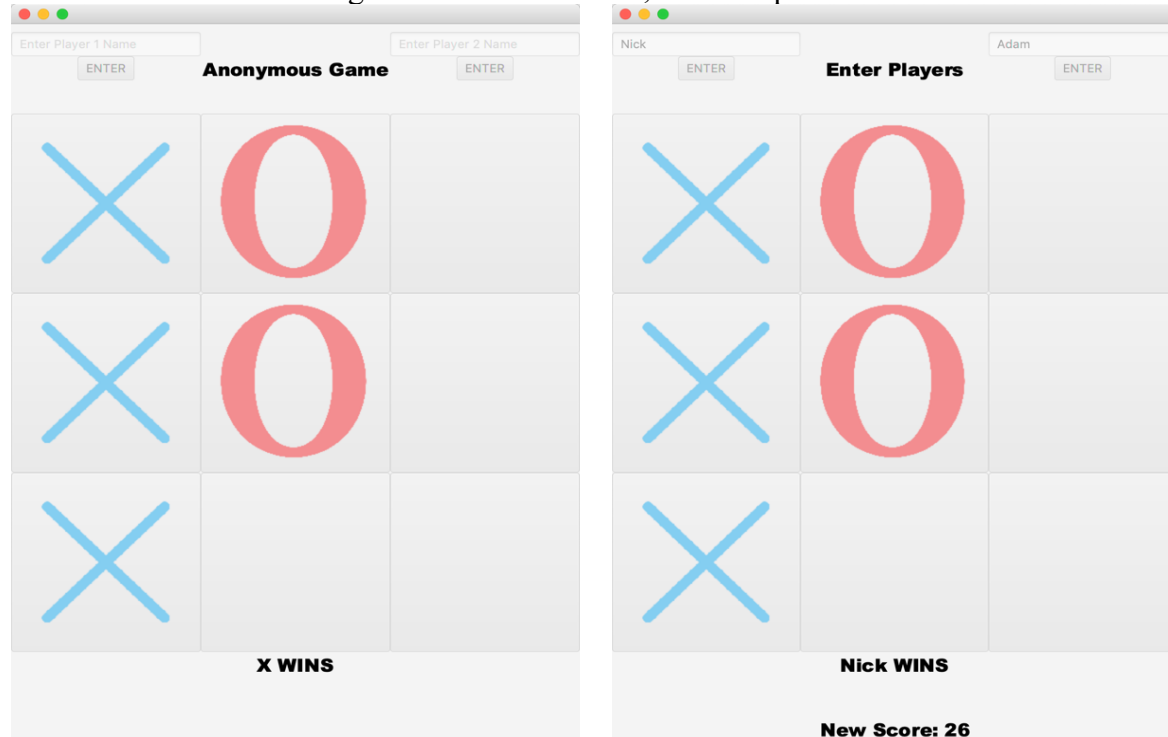


Figure 3 – Winners

When the game is won with real player names, the winning player’s name will appear at the bottom as the winner, for example “Nick WINS” and the row will also show the players new score which is current score +1. This new score is updated in the database so next time the player plays, this score will appear in the first row as shown in Figure 1.

At this point the game is over, there are unclicke buttons but when the game is won, all buttons in the grid become disabled so no more can be clicked.

The TicTacToe.fxml file defines the structure of the UI.

Row 1 and row 5 are made by using VBoxes placed in different grid locations, for example `GridPane.colmindex=1` and `GridPane.rowindex=1` would be the top left of the application. These VBoxes contain the TextFields, Buttons and Text.

Rows 2 to 4 do not use VBoxes, they simply contain Buttons making up a 3x3 game grid using the GridPane locations the same way as row 1 and 5 do.

3.2 Launching the application

Launching the application is done with the main TicTacToe class.

This class extends the Application class from javafx.

It overrides the start method which creates the grid pane from the TicTacToe.fxml file, creates a new scene using the grid pane then puts that scene on the stage and shows the stage.

The main method in the TicTacToe class calls the static method “launch” from the Application superclass. This launches the application.

TicTacToe Application Report

3.3 The Controller

The controller class controls all interactions with the user interface and enforces game logic. The class has a lot of variables that represent the UI components (eg TextFields, Buttons). These variables correspond to fx ids in the TicTacToe.fxml file so we can define what happens to the fields at various points in the game.

There are 3 methods in this class that are called when a button is clicked.

These are;

`getPlayer1Score()`

`getPlayer2Score()`

`mouseClick(MouseEvent mouseEvent)`

This is defined in TicTacToe.fxml using the option “onMouseClicked”

All other methods are called within the class.

`getPlayer1Score()` and `getPlayer2Score()` are called when a player enters their name in the text field at the top of the application and clicks enter.

These methods create instances of the Mysql class by passing the player name as a string. They set the player1 and player2 variables to be the players names entered in the fields.

The method `mouseClick` is the one that controls most of the application.

This is called when a button in the game grid is clicked.

This first checks if it is an anonymous game or not by calling `isAnonymousGame()` which checks if the player1 and player2 variables are set from the above two methods, `isAnonymousGame()` returns true or false.

If it is an anonymous game then the `setupAnonymousGame()` method is called.

This will disable the buttons and fields in row 1, set the instruction to “Anonymous Game” and update the player1 variable to “X” and the player2 variable to “O”.

If it is not an anonymous game, then it will start by setting the players’ names and scores in row 1.

This is done using the Mysql object created in `getPlayer1Score()` and `getPlayer2Score()`. The score is populated using the `getScore()` method from the Mysql class. This score is a running total of games won in the past.

The `mouseClick` method handles players turns. This is done by setting the Boolean variable `playerX` to true or false on each call of the method.

When the game begins, `playerX` is true so when a button is clicked, an X graphic appears on the clicked button, `player2`’s turn appears in the right side of the 5th row, the button clicked is added to an ArrayList of buttons called `buttonsX` so we can keep track of the buttons clicked that show X. The button clicked is disabled and the `player` variable is set to false.

This now means that it’s `player2`’s turn. When `player 2` clicks a button the else part of the conditional statement is triggered. An O graphic will appear, the button will be added to an ArrayList of buttons called `buttonsO` so we can keep track of the buttons clicked that show O, the buttons will be disabled, the text that shows `player2`’s turn will be set to an empty string and `player1`’s turn will be populated on the left side of the 5th row.

On each call of the `mouseClick` method we make a call to the `checkWhoWon()` method. This method checks if there is a winner or if the game is a draw and will stop the game if so.

TicTacToe Application Report

To check if X won the `isXtheWinner()` method is called. This method checks if the `buttonsX` ArrayList contains any of the win conditions such as all buttons in one row, all buttons in one column or all buttons in a diagonal line. There are eight win conditions, each of which are defined in their own methods. One example is `getRow1()` which returns an ArrayList of the buttons located in row 1 of the game grid. If the ArrayList `buttonsX` contains all of the elements of the ArrayList returned by one of these win condition methods then `isXtheWinner()` will return true. `isOtheWinner()` behaves exactly like `isXtheWinner()` but checks if `buttonsO` contains any of the win conditions. Both of these methods return true or false.

If there is a winner then all buttons are disabled by passing true to the `disableAllButtons(Boolean bool)` method. The player turn information is set to an empty string. The winning player is displayed. If the players are not "X" and "O" then it will use the Mysql instance created earlier to update the winning players score with the `updateScore()` method from the Mysql class and will display the new score by calling the `getScore()` method from the Mysql class as seen in Figure 3.

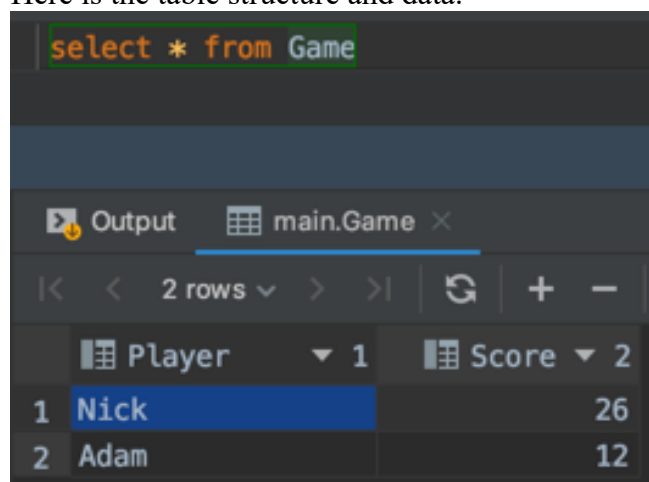
A game is considered a draw when player1 who uses X has clicked 5 buttons without there being a winner since there are 9 buttons and player2 would have clicked 4 already at that point.

This is done by checking that the length of the ArrayList `buttonsX` is 5 and the text in the center of row 5 is still "Game In Progress".

If it is a draw, the player turn information will be set to an empty string, and the textbox field that shows "Game In Progress" is set to "DRAW".

3.4 The Database

To satisfy the requirement of persisting players scores I created a small database that consists of one table called Game which has 2 columns called Player and Score. Here is the table structure and data.



The screenshot shows a database application interface. At the top, a SQL query `select * from Game` is entered. Below the query, there is a tab labeled "main.Game". Underneath the tab, there is a table with two columns: "Player" and "Score". The table contains two rows of data: "Nick" with a score of 26, and "Adam" with a score of 12.

	Player	Score
1	Nick	26
2	Adam	12

Figure 4 – Database

The class that controls the database is called Mysql.

The constructor for the class takes in a String parameter called player which is the player's name for the Player column.

There is a method which gets the connection to the database called `connect()`.

This opens a connection to the `tictactoe.mysql` database running on my localhost.

TicTacToe Application Report

I decided to set up a local database rather than using the one from the lab as I needed to work on it from home mainly and cannot connect to the lab database from outside UCD.

The 2 methods that are called from the Controller Class are `getScore()` and `updateScore()`.

`getScore()` calls `setScoreFromDB()`. This method creates a prepared statement that selects the score where player equals the instance variable "player".

It uses `ResultSet` to hold the results of the query.

If the result set is empty then the instance variable "score" will be set to -1 temporarily.

If the result set is not empty then we get the int at `columnindex1` and set the score variable to that int. `getScore()` then will check if the score variable is set to -1, in this case a new player will be added to the table using the `addNewPlayer()` method. This will create a prepared statement that will insert the player name and a score of 0 into the Game table.

The score variable is also set to 0.

`getScore()` returns the score variable which will be the score for the existing player in the Game table or 0 for a new player.

`updateScore()` is called in the Controller class when a player wins.

This sets the score variable to the current score +1. It then creates a prepared statement which updates the score for the winning player.

There is a `deletePlayer()` method which is used in testing to delete a test player created by `addNewPlayer()`

The database itself was set up as a data source in IntelliJ using the `sqlite-jdbc-3.27.2.1.jar` to connect using java.

4 Class Diagram

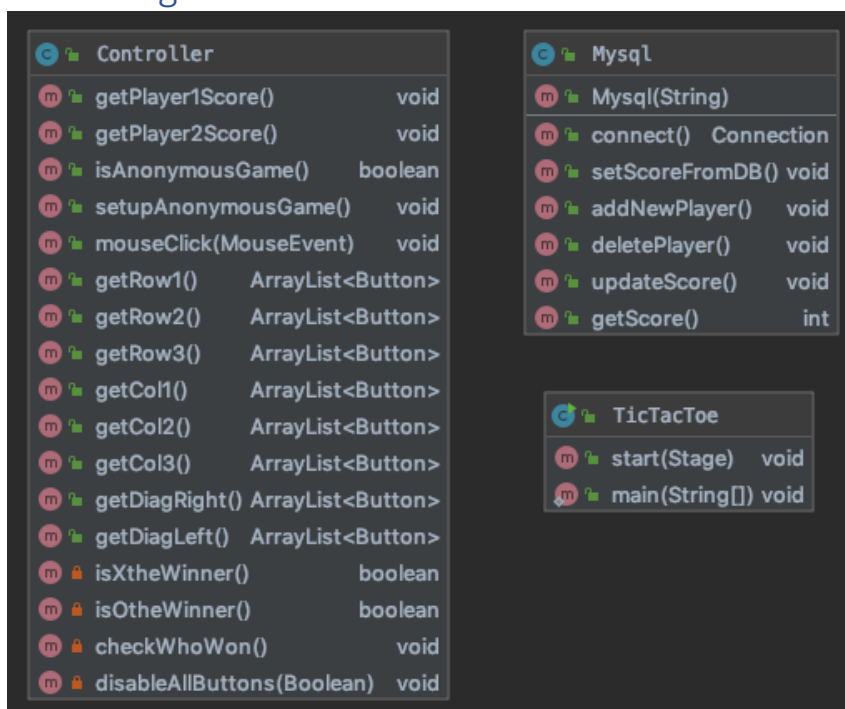


Figure 5 – Class Diagram

TicTacToe Application Report

4.1 Class Design

When starting the project and researching how javaFX applications are built, it became clear that I needed a UI (TicTacToe.fxml), something to control the actions taken in the UI (Controller), something to create the screen and stage and launch the application and something to allow persistence.

The TicTacToe class extends the Application class from javaFX but does not show in the class diagram in IntelliJ.

The only class built in this project that we create an instance of is the Mysql. This is done in the Controller class when setting up both players as explained above.

This is a simple class design for a simple game.

5 Unit Tests

There are 2 tests written for this application.

MysqlTest which tests the Mysql class and TicTacToeTest which tests the controller.

There were 2 strategies used for testing.

The first strategy was to create instances of the class under test to check against variables set and what some methods returned. 10 tests were created using this strategy in TicTacToeTest and 5 were created in MysqlTest

The second strategy was to extend the ApplicationTest class from “org.testfx.framework.junit” along with some other external libraries here.

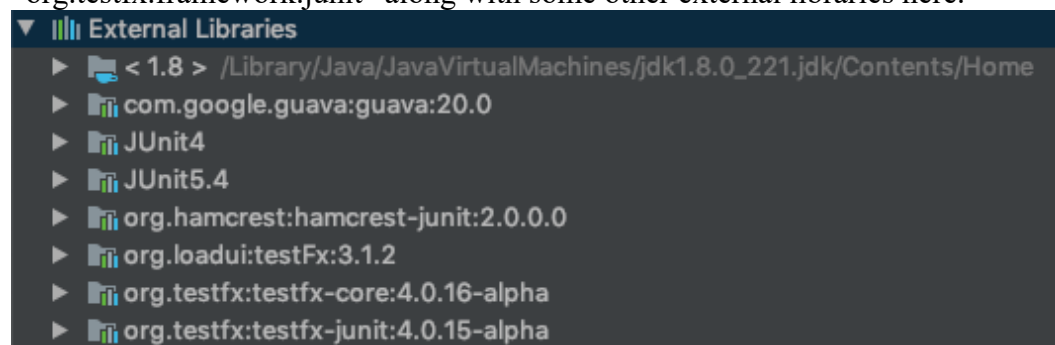


Figure 6 – External Libraries

This allowed me to programmatically click buttons, write text and read text within the application by launching for each test of this type and tearing it down when the test is finished.

The setup to be able to spin up the application and tear it down was found online in the below article. It explains the external libraries needed to do this. Once I had the ability to do this I wrote 21 tests that use this strategy in TicTacToeTest.

<https://medium.com/information-and-technology/test-driven-development-in-javafx-with-testfx-66a84cd561e0>

5.1 Testing the Controller

Both of the above strategies are used to test the Controller.

There are 31 tests for this class so I will not go into the inner workings of each. Rather I will give some tests as an example of how I used both strategies.

TicTacToe Application Report

As mentioned above. The TicTacToeTest class extends ApplicationTest from “org.testfx.framework.junit”. The start method from that class is overridden, it loads the TicTacToe.fxml file, adds that to a scene, sets the scene on the stage then shows the stage. One of the imports for this set of tests is “org.testfx.api.FxToolkit”. This contains an interface called FxRobotInterface.class which has methods such as “write” and “clickOn”. These allow me to have the robot write in fields and click buttons.

This functionality is better demonstrated in the video as part of this project since you can see the robot doing work.

But here is an example of a method that uses this robot and asserts that the application behaves in an expected way after the interaction.

The method setupAnonymousGameTest() tests the functionality of the setupAnonymousGame() method in the Controller class.

This starts by finding the fields we want to test against.

For this I use the find method from the class “org.loadui.testfx.GuiTest”.

The fields we need to find here are instruction, player1name, player2name, player1button and player2button. These are each assigned to a variable.

An example of this is

```
TextField player2name = (TextField) GuiTest.find("#player2name")
```

The field passed to GuiTest.find corresponds to the field name in the Controller and the fxml file.

After finding all of the above fields we want to take an action that would trigger an anonymous game. The action taken is to click on the top left button in the game grid.

The name of this button is #c1r1 which stands for column 1, row 1.

The FxRobot clicks on the button using the clickOn() method from FxRobotInterface.class.

When a game grid button is clicked before entering the player’s names then the anonymous game is triggered. A couple of things should happen. All buttons and text fields in the first row of the application which include the ENTER buttons and the player fields should be disabled so they can’t be used. The instruction text box field should now display “Anonymous Game” rather than “Enter Players”.

So the test asserts that all of these things are true by using the fields we ran GuiTest.find for earlier in the method.

```
assertTrue(player1name.isDisabled());  
assertTrue(player2name.isDisabled());  
assertTrue(player1button.isDisabled());  
assertTrue(player2button.isDisabled());  
assertEquals(instruction.getText(), “Anonymous Game”);
```

There are a lot of tests here that use find(), write() and clickOn() to interact with the game and test the applications reactions to different scenarios. I will talk more about these during the video as it is a lot more visual. I just wanted to explain the libraries used and how I achieved this testing strategy.

The other strategy used is to create an instance of the Controller class to test some of the methods that lent themselves to this type of testing.

An example of one of these tests is getRow1test() which tests the method getRow1().

This created a new instance of Controller.

It creates a new ArrayList of buttons and adds the buttons from the 1st row of the game grid to the ArrayList.

TicTacToe Application Report

We then assert that `getRow1()` returns the same buttons as the buttons in our `ArrayList` from the test method.

This is useful as if something was to change in the future, we need to be sure that our win conditions still contain the correct button locations otherwise they would not be accurate, and the game may behave unexpectedly.

5.2 Testing the database

In all tests of the `Mysql` class I create a new instance of the class with various player name strings. `testconnection()` runs the `connect()` method and assigns the output to a variable, the check is asserting that the variable is not null.

`testScoreIsUpdatedFromDB()` tests that the score variable is set from the database. This test runs the `setScoreFromDB()` method.

If it is set, it will either be 0 or the player's running total score, so the check is asserting that the condition `score >= 0` is true.

`testScoreIsMinus1WhenNameIsNotInDB()` creates an instance with the string "Doesn't Exist". Since that name is not in the databases, the score variable should be set to -1, the check it is asserting that the score is equals to -1.

`testGetScoreForNonExistingPlayer()` runs `getScore` for an instance that has an existing player. The score from the DB for an existing player should be `>= 0` so the check is asserting that `score >= 0`.

`testGetScoreForNonExistingPlayer()` runs `get score` on an instance with a non-existing player called "TESTINGPLAYER". This player should be added to the databases, score should be set to 0. The test asserts that the score is equal to 0. It then uses the `deletePlayer()` function to delete the entry in the DB so it can be tested again.