DAT602 Project Report

Submitted By: Abhimanu Saharan, NMIT ID: 13497164
Submitted To: Todd Chochrane

Recap

Due to the lack of details in Milestone 1, I've included the storyboards with more details, a new CRUD table and the ERD diagram in this report.

Game Description
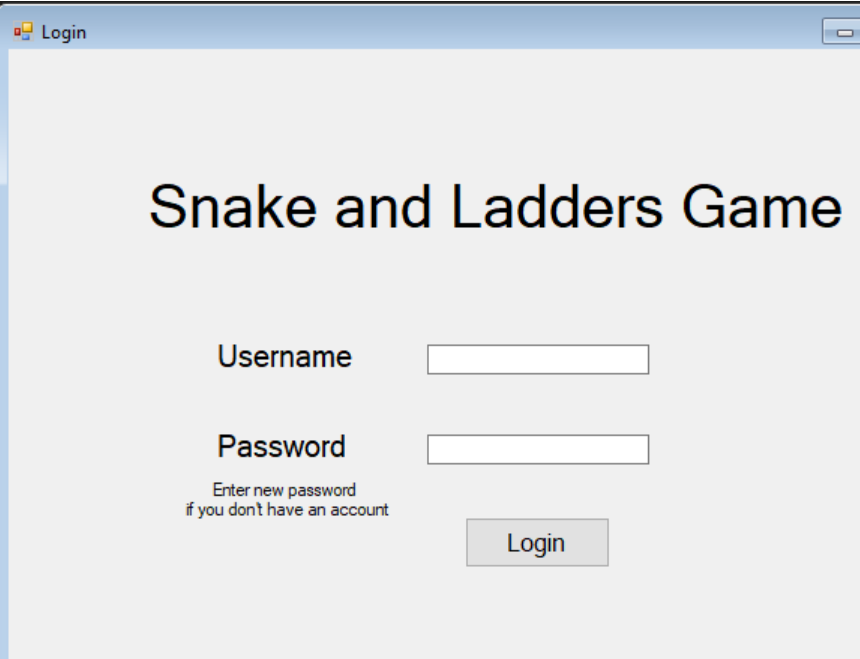

The players roll the dice by clicking on the Dice Button which generates a random number between 1 to 6.

The player's position on the board (10*10 grid) changes by the number generated on rolling the dice. The player who reaches the last grid (10x,10y) first wins the game.

The board has some boxes which have snakes, reaching upon which the player has to go down the defined number of boxes. Similarly few boxes have ladders, upon reaching which, the player can move up the grid.
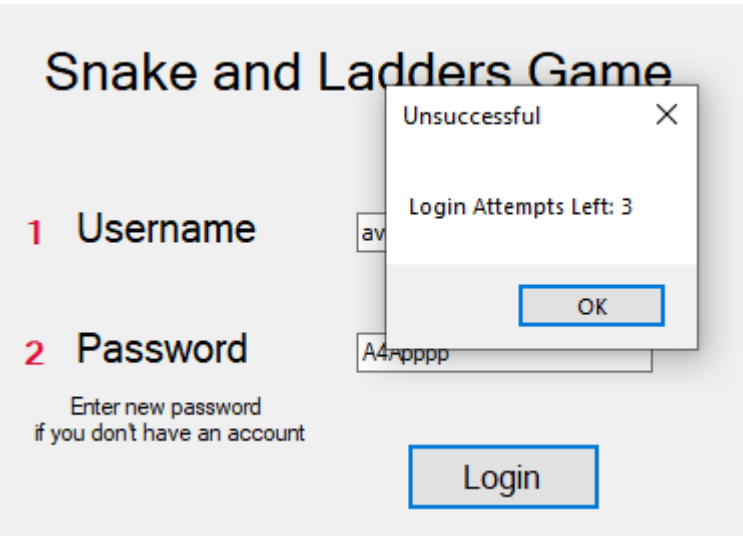
Storyboards

Player Registration

The game starts with the login screen.
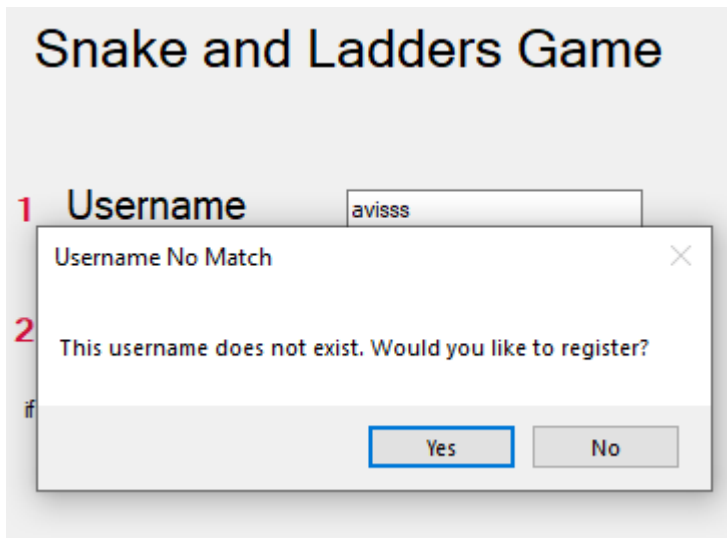If the player is already registered they can login using their password. If their password is wrong the following screen appears:

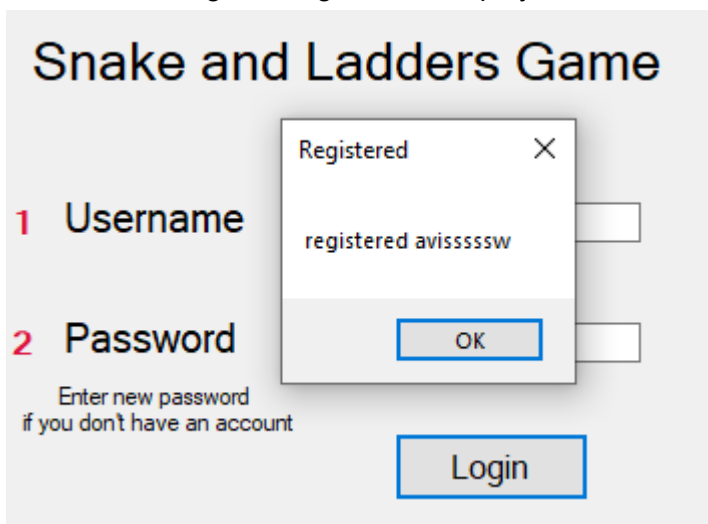If they are a new user the following screen appears:

If the user clicks no, they stay on Login Screen but if they press yes, a new user is created and the following message box is displayed:

After they click OK they are taken to the 'Player Selection' form.

Player Selection/ Game Confirmation

Here the active games (games with one or more online players), along with the players in that game are displayed in two list boxes. On selecting a particular game, the players with the same gameId who are online will be listed in the right list.

1. Join Game: To Join the game with selected index.
2. Create New Game Button: Takes to the 'Create New Game Form'.
3. Admin Panel Link: This will only work if the logged in user is admin. Takes the user to the Admin Form.
4. List of games with one or more online players.
5. List of players in the selected game.

On clicking the 'Admin Panel' button [3], if the user is Admin they will be taken to the "Admin Form" or this message will pop up: If any exception is thrown the message on right will be shown.

On clicking Button 2 the "Create New Game" form appears (on the next page)

Create New Game
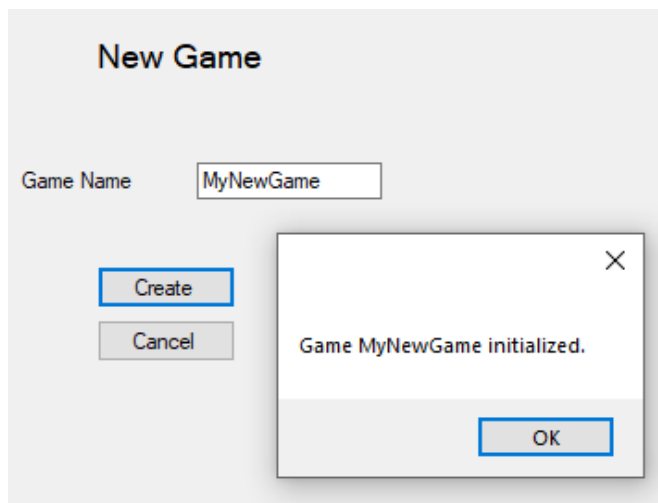
On clicking the create button a new game with no players is created. The players can then join the game.

No two games can have same names, therefore if user enters the name which already exists this error appears:
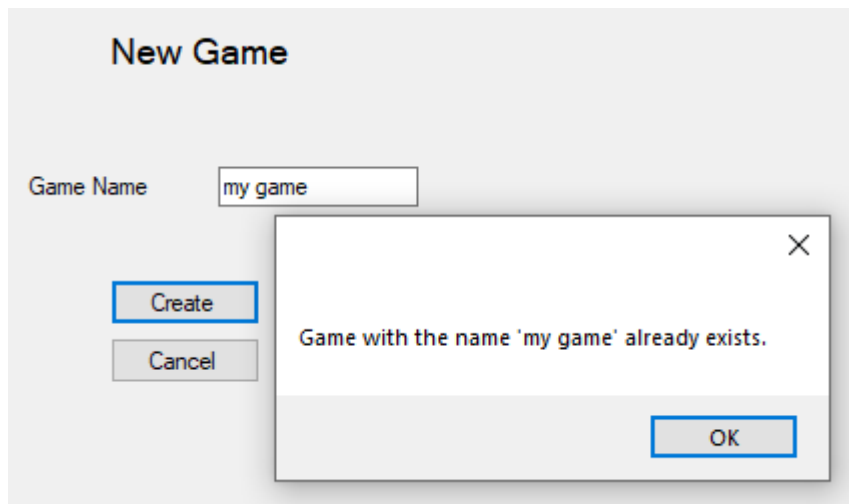
Admin Tools
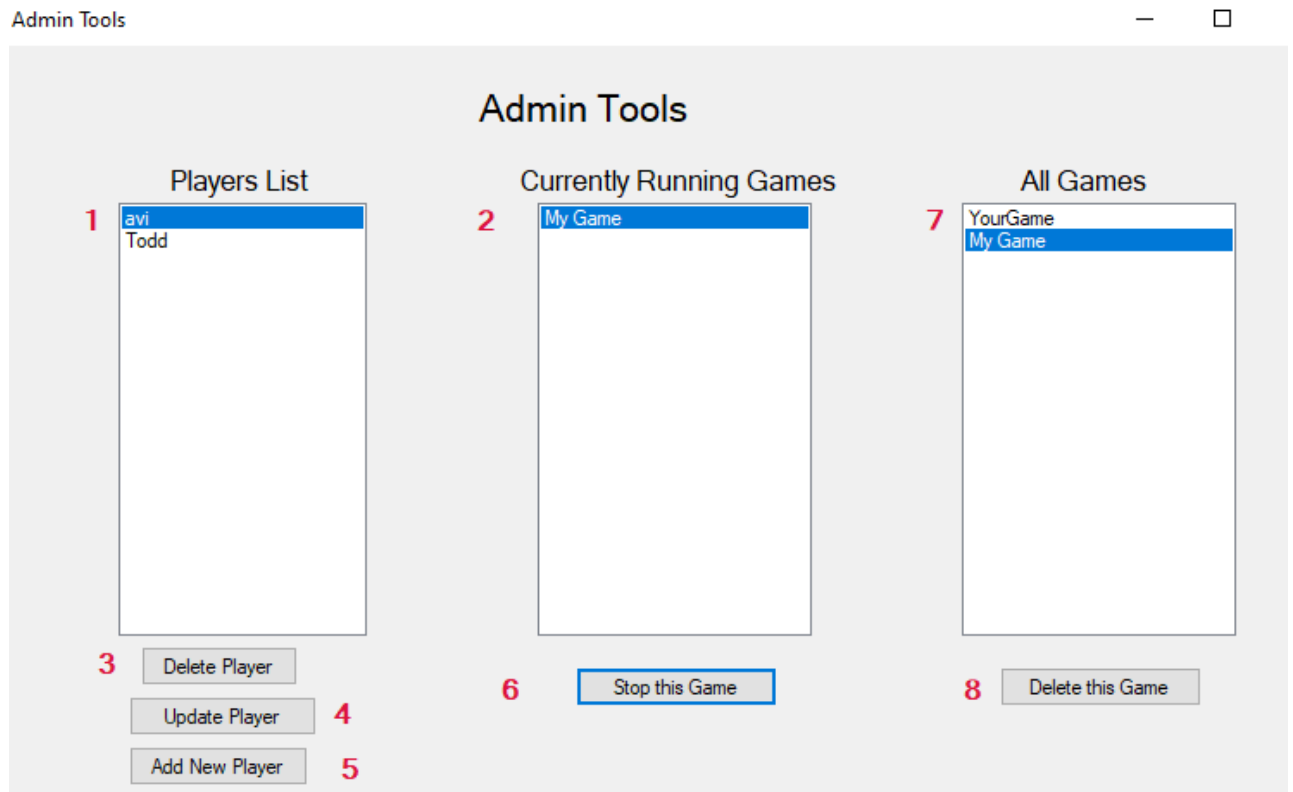
Here the Admin can see the list of all players and can delete them or edit their details. Admin can also delete the games (if any user is playing in that game there active status will be changed to 0 and they will be returned back to login screen)

1. List of usernames of all players.
2. Games with online players.
3. Button to delete selected player
4. Link to the form where the details of the selected player can be updated.
5. Add a new player to database button
6. Button to stop the currently running game.
7. List of all games (Lists 2 and 7 can be combined and a button could be used to filter that list, but I will do that after everything else is done)
8. Delete Button to delete the game. When the game is deleted all players in the game will be logged out, and the game will be deleted.

Clicking 'Update Player' button takes the admin to the 'Update Player Form'

Update Player Details Form

1. New Username Field (prefilled with the username selected from previous screen)
2. New Password field. (The user whose password is changed will remain logged in for the present session, even on changing the password.) To make the player leave the game, either delete player or delete Game could be used.
3. Field to reset login attempts
4. To Edit High Score
5. If 'is Admin' is checked the user will become admin from their next login.

Similarly Delete Button [3] deletes the selected player and they will be logged out.

CRUD TABLE

This CRUD table does not include the CRUD operations for all the procedures exclusively but the CRUD operations of higher level procedures.
Eg: 'Admin- update player details' also includes CRUD operation for Admin unlock user account and Admin-Create new user.

	Check User name	Login	Register	List active games	List active players	Create Game	Join Game	Move Player	Admin-Update User details	Admin Delete Player	Admin Delete Game	Leave Game/ Logout	Move Player	Send Chat Message	Retrieve Chat	Logout
Player	R	R	U		R		R, U	R,U	R,U	D		U	R,U			R,U
PlayerID							R	R	R	D			R			
username	R	R	U		R				R,U	D						R
password		R	U						R,U	D						
High Score									R,U	D						
Is Admin									R,U	D						
Is Online		U			R			R		D		U	R			U
Login Attempts		R,U							R, U	D						
Game Id							U			D		U	R			
location								R, U		D			R,U			
GameSession				R		U		R		U	D	U				

GameID						U		R		R,U	D				
GameName				R		U					D				
gameActive				R						R,U	D	R,U			
Tiles						C									
tileID						C									
xPosition						C									
yPosition						C									
hasSnake						C									
hasLadder						C									
Message														U	R
MessageID														U	R
GameID														U	R
PlayerID														U	R
Message														U	R

The analysis of CRUD operations is after the ERD diagram, in the 'SQL Procedures and CRUD operations' section.

LOGICAL ERD

My ERD is simple. I am using 4 tables-
1. Player table for player details.
2. Game for saving game sessions so that multiple players can join that game session.
3. Tile table just to initialize the coordinates for grids and the tile number representing those grid coordinates. The table 'Tiles' also contain hasSnake and hasLadder fields that are also static for every tileID. If the user steps on any tile where isSnake is 1, they will be pushed back 7 tiles, and the opposite will happen for isLadder.
4. Messages table which has messageID as primary key. (I was thinking of using GameID and PlayerId together as primary keys, but after doing some queries for chat I stuck to messageID because messageID made it easier to store messages incrementally, and also because come player could be deleted by Admin and it could cause inconsistency in chat messages.)
All the tables are described in more details below:

GameSession		
PK	gameID	int
	gameName	VARCHAR(20)
	gameActive	bool

Message		
PK	messageID	int
	message	varchar(300)
FK	GameID	int
FK	PlayerID	int

Player		
PK	id	Type
	username	VARCHAR(20)
	password	VARCHAR(20)
	highScore	int
	isAdmin	bool
	isOnline	bool
	loginAttempts	int
FK	gameID	int
	location	int

Tiles		
PK	tileID	int
	xPosition	int
	yPosition	int
	hasSnake	bool
	hasLadder	bool

Players Table

The players have their unique ids as primary key and have the following other properties
1. username - unique varchar(20)
2. Password
3. High score
4. IsAdmin: To store bool value for if the user is Admin or not
5. IsOnline: To store bool value for if the user is online or not
6. loginAttempts: To store the number of login attempts

7. GameID: This is assigned to every player after they join or create their own game. Players with the same gameID will share the tile map and be able to chat with each other.

GameSession Table

The game session table has the following fields:
1. GameID: Every game session has a separate ID. (Game is same as board)
2. GameActive: This is a derived field whose value is calculated at runtime by checking if any of the players in a Game session are online.If they are, then the value of '1' will be stored in the GameActive variable. I chose to include this field because even though it is a derived value, it is used in several places in the game, and calling a procedure to calculate this every time could slow down the execution of other procedures which depend on this value.
3. GameName: This is the name of a game- provided by the player who will create a new game.

Tile Table

This table will store static data of grids.
1. TileID: This field will contain values from 1 to 100, pointing to the grid position (1,1) to (10,10).
2. positionX: For grid position horizontally
3. positionY: For grid position vertically
4. hasSnake: every tile will have hasSnake value set to either 1 or 0. If the hasSnake value for a tile is true and the user lands on that tile, they will be pushed back 10 tiles.
5. hasLadder: same for the hasLadder field. If this field is true for any tileID and the user lands on that tile, they will move 10 tiles ahead.

Message Table

messageID: Autoincrement unique primary key for every message.
Message: Varchar(300) string. I will use the try catch block to create a message box if more than 300 characters are tried to store.
PlayerID: Foreign key to the player who's sending the message.
GameID: Foreign key to the game in which this message is sent.

SQL Procedures and CRUD Operations:

The SQL procedures for all the CRUD operations are [here](#).
There are more procedures than CRUD table operations because the CRUD analysis contains only the high level procedures, but many procedures/functions which are subqueries or used within other procedures are not included in the CRUD table.

Overview of CRUD Procedures:

CheckUsername: Checks if the passed value is a username in the player Table.

Login: This procedure takes two parameters, pusername and pPassword. Then it calls the CheckUsername procedure to check if the username exists and if it exists then the password is checked against the value of password in the player table where usernames match. If the password is the same, a table with message 'pusername logged in' is selected.
If the username does not match the user a table with message 'Hi! pUsername, would you like to register' is selected. If the password does not match a message saying the same is selected.

Register: Here also username and password are passed as parameters and the values passed are stored into the players table.

ListActiveGames: This procedure is used to query the games which have one or more players online.

CreateGame: This procedure is called when a user wants to create a new game. This does not take any parameter and an entry into gamesession table with incremental gameID is inserted.

JoinGame: This procedure takes pGameID and pPlayerID as parameters and changes the value of isActive field in GameSession table to true where gameID matches to pID and changes the gameID value of player with pPlayerID to match pGameID.

MovePlayer: This procedure takes pPlayerID, pGameID and pDiceValue as parameters and increments the tileID value of player whose playerId equals to pPlayerID and gameID equals to pGameID by pDiceValue.

Admin_Update_User: This procedure takes the pUsername, pPassword, pLoginAttempts, pLoginAttempts and pHighscore as arguments and updates the corresponding the field the the player whose username matches pUserName.

Admin_Delete_Player: This procedure takes pPlayerId as parameter and deletes the row in player tables where ID matches pplayerID.

Admin_Delete_Game: This procedure takes pGameID as parameter and deletes the row where gameID equals pGameID. Because pGameID could be a value of any player's GameID field this deletion also stops their game.

SendChatMessage: This procedure takes pMessage, pUserID, pGameId as parameters and stores these values into a messages table which has an auto incrementing messageID field.

RetrieveChat: This procedure takes pGameID as parameter and selects 10 values sorted in descending order of messageID from message table's message column where gameId is equal to pgameId.

Logout: This procedure takes pUserID as an argument and sets the isOnline value of player which matches ID to pUserID to 0.

ACID

I have used transactions in a procedure where other procedures or other users could modify the data at time when my transactions are run.
Also I've used transactions wherever there were chances of any exceptions getting thrown. Using try catch blocks and transactions I can be assured that no inconsistent or partial data is entered in any table, and I can roll back the transaction to my defined savepoint or completely.

The procedure where the transactions have been most helpful is the login procedure. Here, the username was both updated and retrieved in a series of statements, and the values in those statements (username and password) could be edited by the admins on separate connection, or also new users could be trying to login using the same usernames. The logic for my login procedure was as follows:
1. Look for the submitted username. If the username is found, check the password.
2. If no matching username is found, ask the user if they want to register as a new user.
The problem here was that the admin could delete the existing users or even change their password, while the procedure was executing.
Therefore the use of transaction here is important because if at the end of procedure the username was not updated the transaction can be rolled back and restarted. The default 'Repeatable Read' isolation level was also important here because the uncommitted data from other procedures is not retrieved and the data passed at the start of the procedure can be updated in our own transaction.

My database is reliable even in multiplayer update and retrieval because the data clashing and data loss is prevented using both -logics and transactions wherever required.

MySQL by default fulfills the ACID properties. Eg:
Atomicity: Any BEGIN and END block or UPDATE, INSERT OR DELETE statements if not fully executed are rolled back completely.

Consistency: Referential integrity is maintained by keeping the associated fields linked. Eg: If any field which is the foreign key for other fields can not be deleted until those associated fields are deleted first.

Isolation: The default 'Repeatable Read' isolation level means all the transactions remain isolated until they are committed, i,e- The transactions can not receive data from unfinished transactions.

Durability: This means that the database will keep track of not just completed transactions and data, but also uncommitted or pending transactions. If the connection with the server is disconnected for any reason, those uncommitted transactions will execute on the next connection.

Test.cs

After milestone 1 I started creating the GUI and linked most of my procedures to the GUI. (Except the actual game grid.) Because the GUI is correctly fetching and updating the data, the procedures seem to be working fine.

However as per the requirements of Milestone two, I have created the Test class which runs all the procedures with passed values.

The test data chosen is based on the data that should generally be passed to the procedures, i.e it does include all the edge cases. (Because most of the validation of the data passed by user or returned by procedures could be done by the GUI. Eg: If any procedure includes any exceptions like buffer overflow or incorrect data types passed, the GUI would first prevent passing of that data by the user.

Therefore, these tests only check if the return values of the procedures are correct, based on data that would generally be passed to these procedures.

 The problem I faced during testing is- because the initial namespace for the console app was different from the application in which my DataAccess class (which had methods to fetch data from SQL procedures), I had to include the reference to the DataAccess class in my console app namespace. But even when I did that I had errors that required reference to System.Security.Permissions to pass data between two namespaces. When I solved that, I had an exception thrown by MySql.Data.MySqlClient.Replication.ReplicationManager which I could not understand why. Even copying the data access file to my new namespace did not solve this problem. Therefore finally I just changed my console app's namespace name (which I could have done in the starting.)

Other than that I just had to edit the messages returned by my SQL procedures to provide more details, which would make it easy for testers to understand the test output.

Here is the link to the Test class , and here is the console tests project which contain this class.

Milestone 3 : GUI

All the adjustments made to CRUD, ERD and procedures are as in the above pages.
The design and storyboards here are explained in more detail and include many more game cases than my [initial milestone one](#).

The [procedures](#) were refined to not just manipulate the data but also provide error messages or just normal messages which could be popped up if the data manipulation does not go as planned and the messages could also help the tester understand the output easily, without having to check the database after any procedure is executed.

In the GUI I included forms which the user is redirected to and from, based on the output from procedures when the player clicks any button. I have used message boxes to show different errors, for example when the user tries to access some form which they are not allowed to- eg. admin form.

The forms layout and actions are described in the pages above. Clicking the form name will take you to that forms description:

[Player Registration](#)- Login and Register functions have the same form with different message boxes based on the data that user inputs.

[Game/Player Selection Form](#): This form has buttons which redirect to 'Create Game' form, the admin form (If the user is admin), and the main game screen of the game which is selected.

[Create Game Form](#): This form allow the user to create a new game, which will be listed in the Game Selection Form and then that game could then be joined by other players.

[Admin Panel](#): This form is accessible only to admins. Here the user can edit other players details and even delete the players, or delete/stop any games.

[Update Player Details Form](#): This form is used by admin to update the selected players details and unlock the player if they got locked out by entering 5 wrong passwords.

GUI and database connection

All the forms described above are fully functional and retrieve and write data to the database. The DataAccess class in my project has methods which take parameters from the forms and pass them to the procedures and also store the returned values.
Those methods are then called by the instances of DataAccess class in different classes and forms, whenever an event is triggered in the form (usually when the user clicks any button or selects an item in the list).
[Here](#) is the link to my DataAccess class, and [here](#) the link to the complete project. (The [console tests](#) project is in a separate folder in the github repository.)