Main Program and Subroutines:

Render-Systems: This general system works to render the different parts of the game, it contains a main **Render** method that provides the ability to switch between any of the renderable scenes available in the game.

Render: The Render is the like the “Main” of the LibGDX app. Inside the application loop, the loop strategy is being changed by running whether RenderLogin, RenderGame, or RenderMenu depending on the stage of the application. They work to render the login screen, the single player game and the main menu accordingly. They are individually triggered by a change in the ApplicationStrategy held in the main Render program. This makes it incredibly easy to add transitions as well as to facilitate those that are already in place.

The renderStrategies use subroutines to provide for the implementation of the game. These are as follows:

Database-Subroutines: The database subroutines are included in a set of classes that extend the abstract class DatabaseController, which contains the basic objects related to database interactions (Connection, PreparedStatement, ResultSet) and the method to close the connections. The database subroutines are used by the Login and Game renderer in order to keep track of the game play information as well as create, validate and store the users account information.

Authentication: The AuthenticationController provides the RenderLogin with the ability to validate user credentials.
Registration: The RegistrationController allows new users to be registered through RenderLogin.
Score: The ScoreController provides the ability for match details to be recorded and retrieved for users.
Leaderboard: This LeaderboardController simply provides access to all the scores in descending order.

Game-Functionality-Subroutines: This set of Subroutine components provide the ability for certain game play attributes to be present
.

Puck-Pusher-Scoreboard: These 3 classes interact with each other in order to bring the Air Hockey game to life. The ScoreBoard is just used to keep track of the score. The Puck and the Pusher are game objects with x and y coordinates, that move around the screen (as they are rendered). The Pusher position is modified depending on the keyboard controls, while the Puck has some movement speed variables (deltaX and deltaY) which define its movement on the screen, since it has to move independently after it was hit by the Puck. The Pusher checks if it touches the puck and if it does the movement speeds of the Puck will be changed accordingly in order to simulate a collision.

Besides the collision with other objects, the Puck uses the PuckState() interface in order to define its behaviour depending on its position on screen. If the Puck is outside the game range then it will change its movement variables when it collides with the edges of the screen, but when it is inside the gate range, the Puck will be able to go past the edge of the screen in order to score a goal. This behaviour is specified in the behaviour function of the specific State.

Math-Utils: Math Utils is a small collection of methods that are used by the Pusher and the Puck in order to calculate the euclidean distance between the objects and detect if they overlap in order to execute a collision action.

Drawing-Subroutines: We have also created some subroutines like drawGameObject() or drawText() to ease the task of drawing different elements on the screen.
Menu-Subroutines: The subroutines of the Menu mainly set the right buttons into their right positions and check if any hover or click action has been registered on these buttons.
Login-Subroutines: The login uses registration as well as authentication subroutines in order to register new users and authenticate users that are already in the system. This is achieved by interfacing with the database subroutines.

Motivation: We chose the Main Program and Subroutines architecture due to its relative simplicity and fit for our game. The modularity allowing us to switch between renderers was crucial in keeping redundancy to a minimum. The disadvantages usually seen with this architecture are that it doesn't scale well and data can be changed across subroutines. Since Air Hockey is a relatively simple game the first point became an advantage for us, in our case, trying to implement a much more sophisticated architecture would have been overkill. Also, individual subroutines don’t have the ability or need to modify the same data across subroutines, so in regards to our program this was a non-issue.