Group 49 Asteroids

Ji Darwish

Zohar Cochavi

Moshiur Rahman

Thijs Nulle

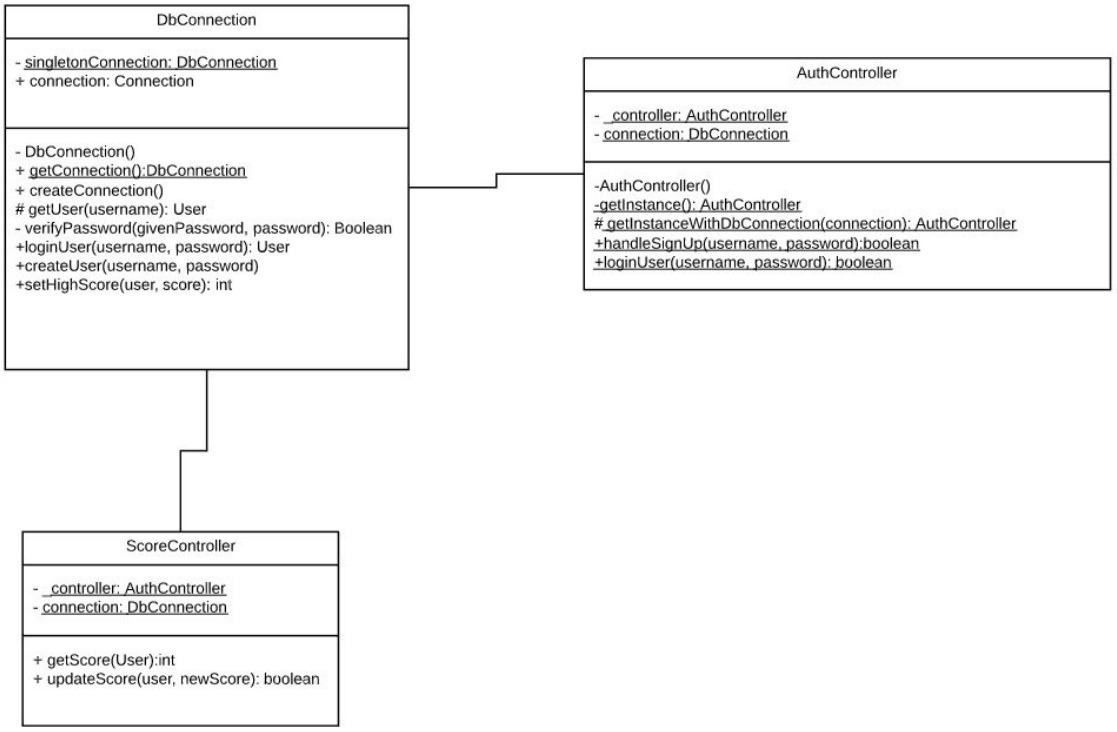
Adrian Kuiper

Exercise 1 - Design patterns

Singleton Design pattern

We used the Singleton design pattern with the DbConnection class to guarantee that only one connection existed at a time. It also ensures the integrity of the database because there's only one connection modifying the data. Also since we made this into a singleton because then we wouldn't have any static methods but instead, instance methods which are way easier to test.

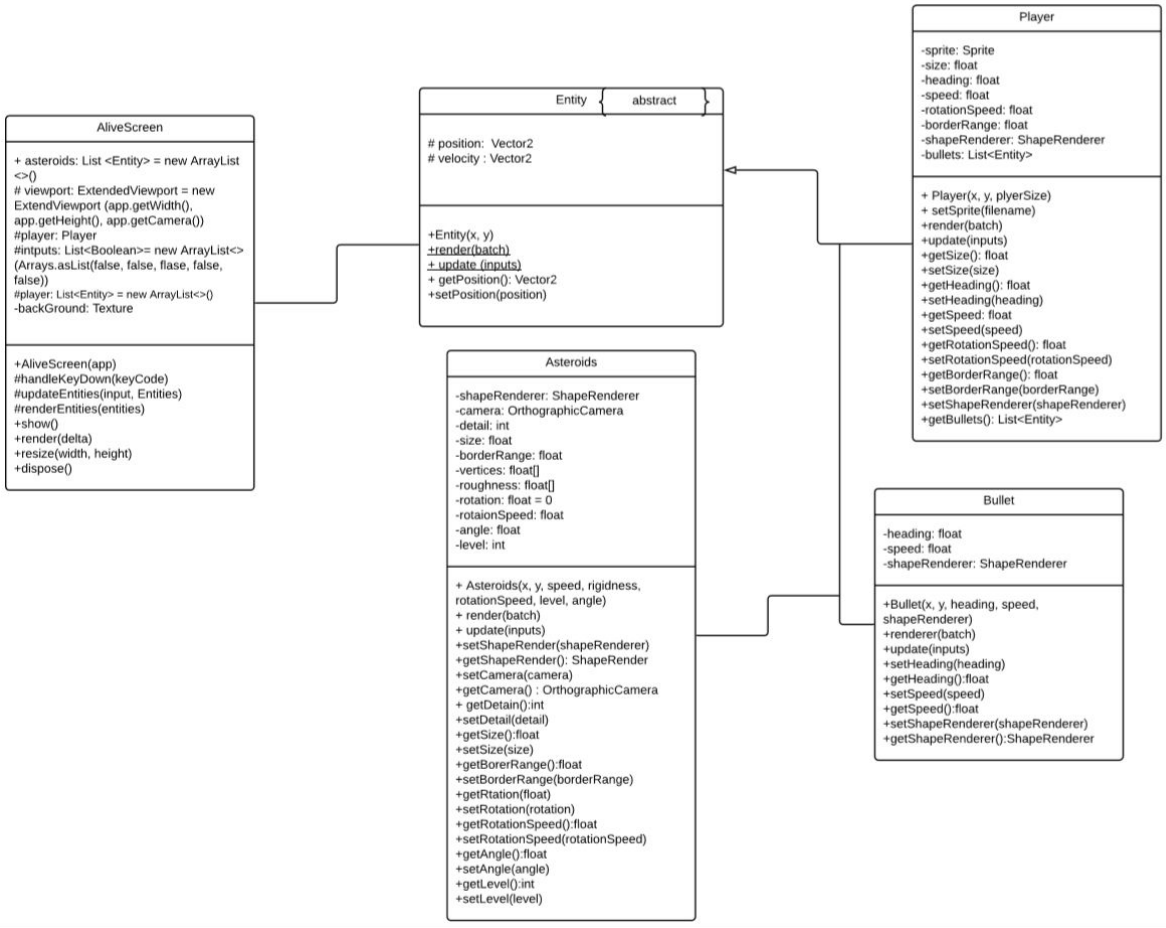
The only way to get access to the connection is through a static method called getConnection() in the DbConenction class, which provides a global access point to the connection. The getConnection() method checks whether there exists a connection to the Database in the system already if the answer is yes, it just returns the connection, but if there exists no connection, then the method calls the createConnection() method (creating a new connection) and then returns it.



Template Method

We noticed that all entities in our game have similar behaviour in regards to rendering and updating. So we wanted to centralize the shared behaviour and make it possible to extend it for more customization of each subclass of an entity.

We defined an abstract class (Entity) which implements the basic behaviour of rendering and updating. Then we went on creating all entities in our game (Player, Bullet, Asteroid) and wherever possible we just used the default behaviour and in some other cases, we were able to extend the default behaviour to customize it more for different types of entities.



Exercise 2 - Software Architecture

We have decided to make use of the Model View Controller architecture as it is perfect to design our game using this architecture. For our Game, asteroids, we make use of a graphical user interface, which is our view, where the user can interact with the application or play the game by clicking the buttons on the GUI or use the keyboard when playing. All the data about the user and the high scores are gathered in the Database which functions as our model. When the user makes an interaction with the GUI these interactions are handled by our controller, which also controls requests made by our user interface for receiving data from the database. These three components are essential for our game design and the reason why we chose to use the Model View Controller architecture.

Model:

For our model, we made use of the MySQL server provided to us by the TU Delft. Through JDBC driver we let the program interact with the database, in which we made tables that store the player's username and password and store the high scores made by the player. Also, the Database provides security against attackers who want to hack passwords using SQL injection, by making use of prepared statements and hashing passwords to prevent these kinds of malicious statements. Most of our logic also lies in this area. The classes under the model are DatabaseConfiguration, User and DatabaseConnection.

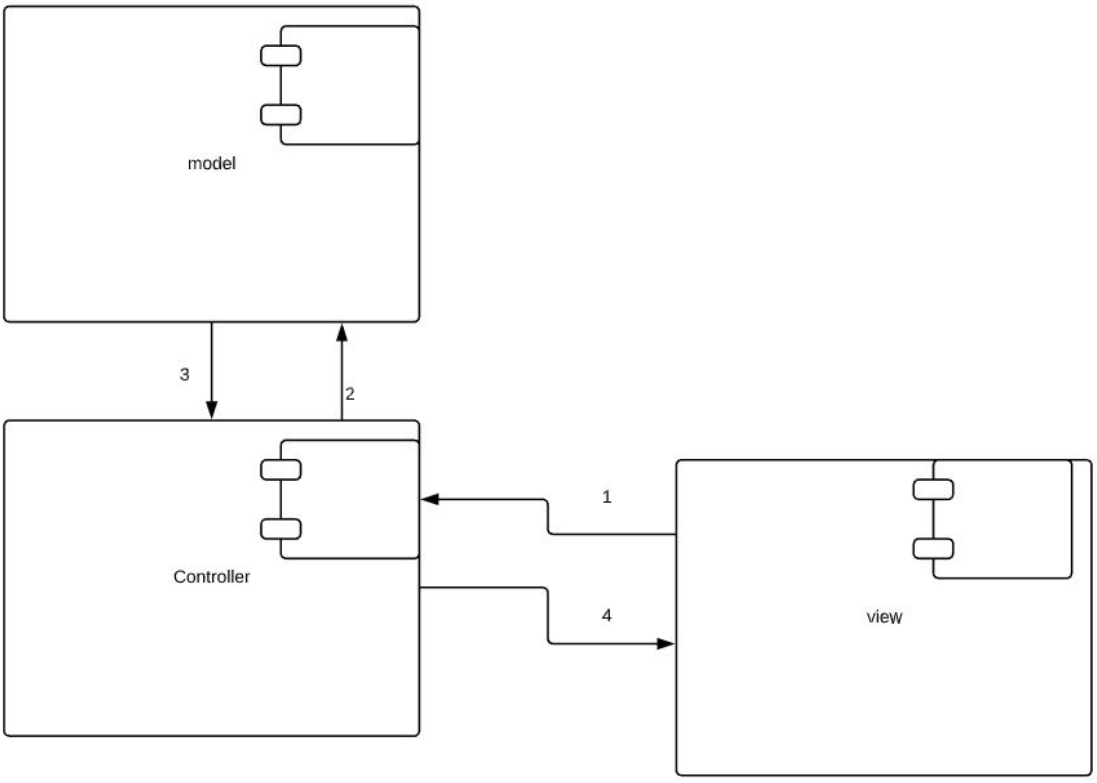
View:

For our view we have a few different screens:

- Login Screen
- Signup Screen
- Main menu screen
- Game screen
- High Score Screen

The user can interact with these screens via the controller. For example by logging in, creating a username or just playing the game. These classes are what the player sees. These are essentially the GUI of the game. Moreover, the depiction of data coming from the model is shown in these classes. Everything under the view is what the user sees. The user can log in, signup, see the main menu, play the game and finally view the high score.

Controller:
Our controller classes are AuthController and ScoreController. The controller lies between the model and the view, ensuring that whenever the view wants to ask something from the database it calls a function on the controller. This helps to separate concerns so that it's easier to work on specific tasks, Also, it ensures that nothing goes wrong due to some misused function from the view. The controller also has connections to the model. It acts as a bridge between the view and the model. Whenever the user clicks something the view or makes a request in the GUI, it comes here and then goes to the model.



There were many reasons why we chose this architecture pattern over the other. The primary reason was that the other ones are incompatible. Initially, we thought of using the client and server model but that wouldn't work with our game because we don't have a server. After that, we thought of using the ports and server, but that didn't work either because our business logic was too integrated from the other services. Like this way, we tried many patterns but in the end, we chose the MVC because it worked best with our game