Design patterns

Template method pattern

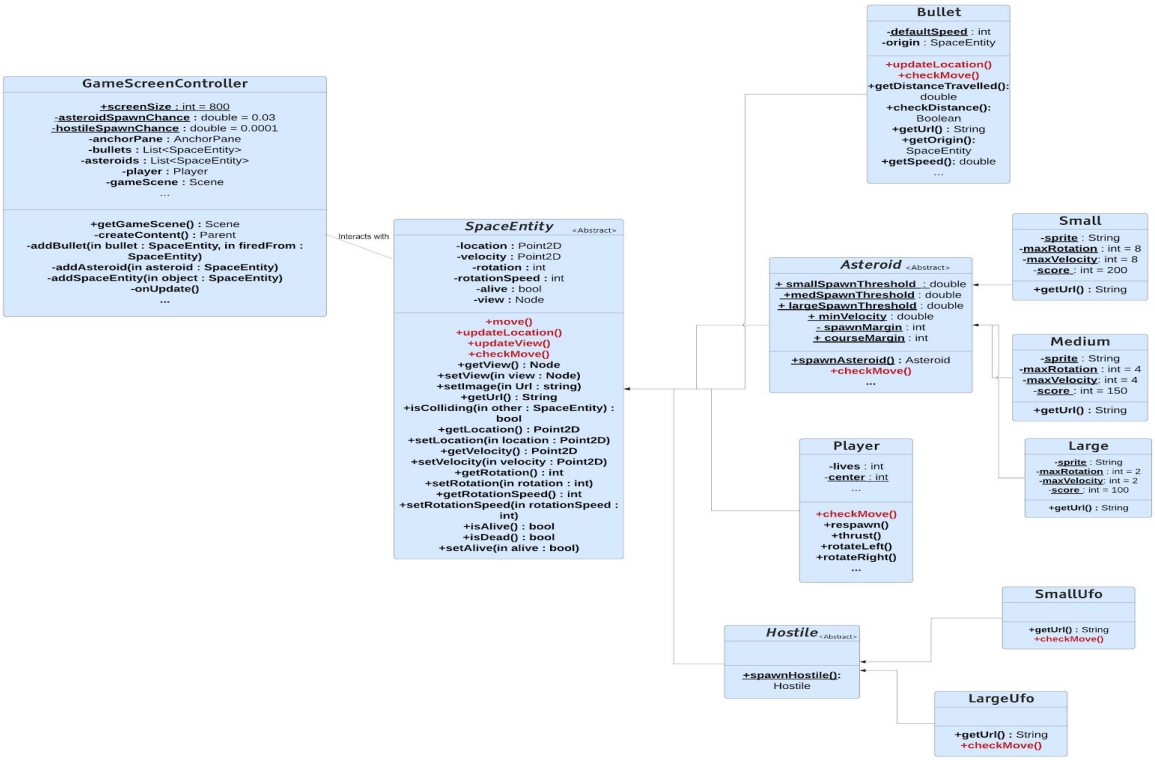
1. We have used this pattern for implementing the SpaceEntities, Asteroids and Hostiles and the method that deals with their movements.

We have chosen this pattern to localize common behaviour of the subclasses and keep it in an abstract class in order to avoid code duplication and to make extending the subclasses or adding a new type of space entity much easier. Additionally, we have implemented the invariant parts of the moving algorithm once, in the base class (SpaceEntity), and left it to the subclasses to implement the varying behaviour.

We have made an abstract class, SpaceEntity, which has all the methods and attributes used/needed by all the inheriting subclasses: Player, Bullet, Asteroid and Hostile. These classes further implement methods and attributes specific to their behaviour.
More specifically, the *move* method in SpaceEntity makes use of this pattern in the following way: it now calls 3 separate methods :

- *updateLocation*, which has an implementation in SpaceEntity but can be overridden by child classes
- *updateView*, which is implemented in SpaceEntity but can't be overridden (as there is no difference in this behaviour of the subclasses)
- *checkMove*, which is not implemented in SpaceEntity and has to be overridden by child

2. Class diagram: the aforementioned methods are highlighted in red



3.

-The *move* method in the base (SpaceEntity) class, calling the 3 separate functions and their implementations in the base class.

```
public abstract class SpaceEntity {
    ...
    left out code
    ...

    /**
     * Function that updates the location and rotation of the spaceEntity,
     * to be called every frame.
     * Cannot be overridden.
     */
    @Generated(message = "")
    public final void move() {
        updateLocation();
        updateView();
        checkMove();
    }

    /**
     * Standard location update method.
     * Can be overwritten by child classes.
     */
    public void updateLocation() {
        setLocation(getLocation().add(getVelocity()));
        setRotation(getRotation() + getRotationSpeed());
    }

    /**
     * helper function of move, which updates the view of the spaceEntity.
     */
    public final void updateView() {
        getView().setTranslateX(getLocation().getX());
        getView().setTranslateY(getLocation().getY());
        getView().setRotate(getRotation());
    }

    /**
     * A function to check if the new position of the spaceEntity is valid.
     */
    public abstract void checkMove();

    ...
    left out code
    ...
}
```

-Example of a subclass (Bullet) overriding *updateLocation* and implements *checkMove*

```
public class Bullet extends SpaceEntity {

    ...
    left out code
```

```
    ...

    /**
     * Updates location of the Bullet.
     * Used in the #Move method
     */
    @Override
    public void updateLocation() {
        Point2D oldLoc = this.getLocation();
        setLocation(oldLoc.add(getVelocity()));
        setRotation(getRotation() + getRotationSpeed());

        Point2D newLoc = this.getLocation();
        this.distanceTravelled += oldLoc.distance(newLoc);
    }

    /**
     * Checks if the Bullet move is valid.
     * Wraps bullet around if it goes out of screen
     * Used in the #Move method
     */
    @Override
    public void checkMove() {
        checkWrapAround();
    }
    ...
    left out code
    ...
}
```

Facade pattern

1. We have used this pattern for decoupling the database form the rest of the project.

We have chosen this pattern to simplify the interface of communication with a subsystem, in our case the database.

We have made a Database class that acts as a facade. It provides a number of methods for communicating with the SQL database, for example to establish a connection and to query the data. This way the client doesn't have to use SQL statements or commands, they just use the simple Java code provided by the Database facade. One example is that when it comes to queries, the client does not have to write an actual SQL query, they just have to call the appropriate method in Database.

Some of the methods that our facade provides are:

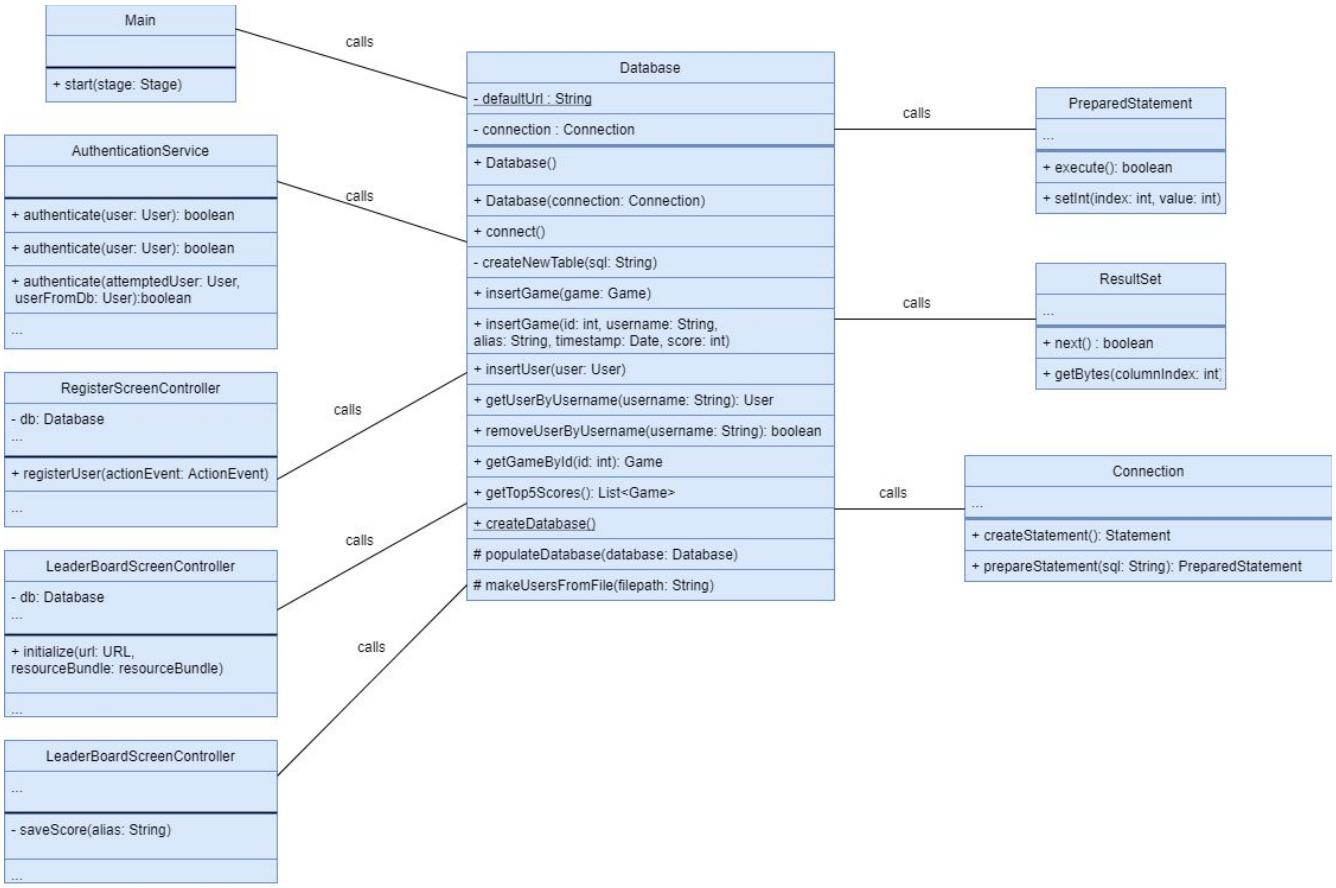
- *createDatabase*, which establishes a connection to the database
- *getTop5Scores*, which performs a query
- *insertUser*, which performs an operation

Some of the SQL components that our facade calls are:

- java.sql.PreparedStatement
- java.sql.ResultSet

- java.sql.Connection

2. Class diagram:



3.

-Examples of Database methods masking database specific operation

```java
/**
* Inserts a record into the user table.
*
* @param user the User that will be added to the database
*/
public void insertUser(User user) {

   try (PreparedStatement statement = connection
           .prepareStatement("insert into user values(?,?,?)")) {

       statement.setString(1, user.getUsername());
       statement.setBytes(2, user.getPassword());
       statement.setBytes(3, user.getSalt());

       statement.execute();
   } catch (SQLException e) {
       System.out.println("error when inserting user, user"
               + "already in database or connection could not be established: "
               + e.getMessage());
   }
}

/**
```

```java
* Gets the Games with the top 5 highscores.
* @return ArrayList containing the top 5 games.
*/
public ArrayList<Game> getTop5Scores() {
   ArrayList<Game> highScores = new ArrayList<>();
   try (PreparedStatement statement = connection.prepareStatement("select "
           + "* from game order by score desc limit 5")) {

       ResultSet resultSet = statement.executeQuery();

       while (resultSet.next()) {
           int score = resultSet.getInt("score");
           String username = resultSet.getString("username");
           String alias = resultSet.getString("alias");
           Date timestamp = resultSet.getDate("timestamp");
           int id = resultSet.getInt("id");

           Game game = new Game(id, username, alias, timestamp, score);

           highScores.add(game);
       }

       resultSet.close();

   } catch (SQLException e) {
       System.out.print("error: connection couldn't be established\n");
   }

   return highScores;
}
```