Project Status, Group 6:
CMPU-203

Our project was the creation of a duck dating app. This apps goal was to create something similar to Tinder, with dating profiles for the ducks from the original duck app being compared to each other to create a compatibility score. This score will then help the user make decisions in a GUI interface with options based around the ducks, mostly related to dating app options such as matching with others and seeing previous matches.

In order to do this we needed to create a number of things the ducks would have a preference for in each other to use to create a compatibility score. We decided upon using color, size, a letter, and the type (based on the classes) of the other duck. For this we only had to add a color to the duck, sine size, and type already existed, and prefered letter was based off of the duck's names. To add color Ryan created a random color generator class, ColorGenerator. The result was stored in the variable color, of type Color.

All of the ducks preferences were stored in the variable profile, of type Profile. The profile class stores an int[] typePref which holds integers from 1 - 100 representing the percent preference for each type of duck. There are also int[] sizePref and colorPref, which work the same way, but for duck sizes and colors respectively. It also holds a char letPref, which is the ducks favorite letter. With the exception of letPref, which is randomly generated in the Profile class, all of these variables have a class which is used to generate those scores. Each of these classes (SizePrefGen, TypePrefGen, and SizePrefGen) implements the PrefGen interface, which causes them to need a prefScores function. In all three cases prefScores uses a slightly different for loop and random number generator to create int[] to be assigned to the corresponding variable in the profile.

These profiles are used to create a compatibility score. Compatibility scores are created in the Compatibility class, which holds a double score, two AquaticBirds b1 and b2, a Profile p, a static String[] types, a static Integer[] sizes, and a static Color[] colors. Inside of Compatibility is a Compatibility constructor which creates the score,a calculate function which creates an integer for each type of preference used in calculating the Compatibility, a letPref function which checks if the other duck has the prefered first letter, a toString function which returns a string representing how compatible the two ducks are, and a score function which returns the actual score.

The final part we managed to implement was some amount of user interaction. In this the user first creates a duck, and can then decide to try to match with another duck, see previous matches, or exit the program. All user input works using a reader. To do this we used a state pattern with the class Session being the base, and the classes Choice, Options, History, Intro, and EndState extending the abstract class State which has a constructor and a function run to be overridden. Inside of a session the user can from state to state. The intro state creates the users duck, which is the first duck for all future uses that require ducks. This state is the first state, and cannot be returned to. In between all other states the user goes to the Options state. From here the user can input a 1 to go to the Choice state, a 2 to go to the history state, and a 3 to go to the end state. In the choice state a random duck is created, and most of its information (name, size, compatibility score and type) is shown. The compatibility score takes the users

duck, the randomly created duck, and creates a compatibility between them. Based off of this the user can input Y or N to try and match with it. Then, based off of the score there is a chance the other duck will choose to match with the user too. After this it will return to the options state. The history state has no user input, but just returns a list of all previous matches before returning to the options state. The endState will allow the user to quit after warning them no data is saved. The user can input Y or N, and on a Y the program will quit, but on a N it will return to the options state.

Everything up until this point we managed to get to work. Due to finding a good stopping point there isn't anything we really have at an "in-progress" point, but we were working on adding all of our tests which we did not manage to finish even for what we have done. We have a class diagram and an interactions diagram in our github repository to be looked at. The interactions diagram only has interactions relevant to the dating app portion, while the class diagram is all classes.

After this we wanted to add a proper GUI. The intro state would have drop down menus (type, color, etc.) and boxes the user can type in (name, letter, etc.), more options the user can pick for their duck (adding color, favorite letter, etc.), and a button to complete creation. For the choice state his would include showing a random picture of a duck, with a border of the color of the duck. All preference information and compatibility information would also be present here. Instead of typing to try to match, we would have some sort of button or slider system (with moving left or right saying yes or no) allowing the user to try and match or not. The history state would still be a list, but there would be an added button the user could click to go back to the options state. The final state we have implemented, endState, would show a warning message about not saving, and then a yes or no button to proceed with quitting.

After that we would like to try to add geese as another matching option. This would be pretty simple, but require updating array sizes and possibilities for the goose class.

We would also like to add some sort of save feature, allowing the user's duck to save previous matches and their own information. This would probably require some sort of file system, with different files holding the profiles for your duck, and each of the other ducks.

It could also be fun to increase the "social" aspects of the app. We could add some inter-duck messaging system, and a social group system. The messaging system could be implemented by letting the user go to the history state, selecting a duck, and typing a message into a box and then having that message "sent". This would probably happen in a new state. Of course, since ducks can only quack, we could easily create a random generator to have a random type of quack (no quack, multiple quacks, a single quack, etc.) returned. This would probably require changes to some of the quack related classes as most of the messages are not currently conducive to this idea. There would be the option to continue the conversation of leave back to the history state. Most likely the conversations would not be saved. The duck social groups would allow for ducks to have friends. This would probably require creating a list of for each duck holding the reference to other ducks, or their unique IDs, so these friends can be identified by other friends looking at that ducks friends. Using this we could have the ducks with a mutual friend in the choice state to have a greater compatibility score. This would require modifying the compatibility class and making the social groups part of AquaticBird. We could maybe use the observer pattern in some way to look at who is friends with who.