Name: Marc Luzuriaga

UID: 205916585

Project Name: Project 5

**Notable Obstacles**

The biggest obstacle that I faced in this project was deciding how to implement the FrequentFlyerAccount class. Particularly, I found it difficult to connect the PlaneFlight class to the freeFlight function. At first, I had thought that I had to build the function all over again within the freeFlight function. However, after looking back at my notes, I saw that creating a class represented creating a variable. Therefore, I, instead, used PlaneFlight as a parameter that was passed into the function and used to function method calls to build the function.

Another obstacle that I overcame was figuring out control flow within the individual functions to not pass invalid inputs when ininitalizing a class object. I found this difficult because I had thought that you needed to cout something to let the user know that their inputs were invalid. It turns out that all you needed to do was use an if statement to check if the parameter was valid to be passed into the initialized object. Otherwise, it should not get passed in.

**Test Cases**

1.

//Test Case 7

PlaneFlight shortleg1("Howard", "LAX", "LAS", 49.00, 0);

PlaneFlight longleg1("Howard", "LAS", "NYC", 399.00, 2800);

PlaneFlight sample1("Sample", "Sample", "Sample", 0, 1);

FrequentFlyerAccount account1("Howard");

assert(shortleg1.getFromCity() == "LAX");

assert(shortleg1.getToCity() == "LAS");

assert(shortleg1.getName() == "Howard");

assert(to\_string(shortleg1.getCost()) == "49.000000");

assert(to\_string(shortleg1.getMileage()) == "-1.000000");

// account balance starts at zero...

assert(std::to\_string(account1.getBalance()) == "0.000000");

assert(account1.getName() == "Howard");

assert(account1.canEarnFreeFlight(3300.00) == false);

// flights add to an account balance

assert(account1.addFlightToAccount(shortleg1) == true); // returns true because the names match

assert(account1.addFlightToAccount(longleg1) == true); // returns true because the names match

assert(std::to\_string(account1.getBalance()) == "2799.000000");

// free flights reduce an account balance

if (account1.canEarnFreeFlight(285))

{

assert(account1.freeFlight("LAS", "LAX", 285, sample1) == true);

// Howard earned a free flight...

assert(sample1.getName() == "Howard");

assert(std::to\_string(sample1.getCost()) == "0.000000");

assert(sample1.getFromCity() == "LAS");

assert(sample1.getToCity() == "LAX");

assert(std::to\_string(sample1.getMileage()) == "285.000000");

// account has been reduced for this free flight...

assert(std::to\_string(account1.getBalance()) == "2514.000000");

}

else

{

assert(false); // there are enough miles in the account...

}

Although this test case looks similar to the one provided in the specs, it is nto the same. I changed the values to check if my program handled invalid inputs. Starting from the top, I changed one of the objects to a mileage of 0 to check if the program converted the mileage to -1. Next, I had to check if the new balance would property calculate throughout the program

2.

PlaneFlight f1("Howard", "LAX", "LAS", 65.00, 285);

f1.setToCity("LAX");

f1.setFromCity("LAS");

assert(f1.getToCity() == "LAS");

assert(f1.getFromCity() == "LAX");

I used this test case to test if the program caught the invalid input of whether the city’s were being set the same. In this case, the program was able to catch it.

3.

PlaneFlight f3("", "LAX", "JFK", 1000.43, 4);

f3.setName("Marc");

f3.setCost(-1);

f3.setMileage(0);

assert(f3.getFromCity() == "LAX");

assert(f3.getToCity() == "JFK");

assert(f3.getCost() == -1);

assert(f3.getName() == "Marc");

I used this test case to ensure that the mutator functions were correctly modifying the instance object and setting invalid inputs for the mileage and cost. In this case, the mtuators did work and was able to set the empty name to “Marc,” the cost to -1, the mileage to -1.