Eideen Mozaffari

UID: 105988436

Project 5 Report: FrequentFlyerAccount

***Notable Obstacles/Challenges***

1. Understanding classes/objects and the differences between them
   1. It was my first-time using object-oriented programming, so naturally I initially struggled to understand how we were supposed to construct an object, modify its member variables using member methods, and how we could call the public member methods of one class in another class. After it finally clicked, the project seemed a lot more straight forward and less overwhelming.
2. freeFlight
   1. The freeFlight function was the hardest one for me to understand when I first saw it because I didn’t know how I could modify an object of another class using a member method of this class. However, once I realized how we can pass the parameters of a member method of one class to construct an object in another class, I understood the intuition behind why OOP is so powerful.

***Test Data***

1. I tested using the given asserts in the spec, but I will not copy and paste those here. Instead, I will list some of the things that the spec’s asserts tested on:
   1. Creating multiple objects of the PlaneFlight class and creating an object of the FrequentFlyerAccount class.
   2. Using the member methods of the PlaneFlight class.
      1. Using the setters in the PlaneFlight class to initialize an object instance.
      2. Using the accessor methods in the PlaneFlight class to access private member variables in “main.”
   3. Confirming the initial balance of the FrequentFlyerAccount object is 0.
   4. Using an accessor method of the FrequentFlyerAccount to get the “name” associated with the object.
   5. Calling addFlightToAccount on a valid flight to check how the account mileage changes.
   6. Calling canEarnFreeFlight to see if a FrequentFlyerAccount object has enough milage balance to cover a flight of x miles.
   7. Calling freeFlight and passing a PlaneFlight object by reference (valid parameters), and checking new values ***in the PlaneFlight Object***
      1. Check the new to/from
      2. Check the new name
      3. Check the new cost
      4. Check the new mileage
      5. Check the new mileage balance in the FrequentFlyer Account
   8. Pass in non-matching names to addFlightToAccount to make sure the mileage balance does not get affected
2. Now, I will copy and paste the asserts I wrote that test the code and accounted for some edge cases:

// Given PlaneFlight Objects

* 1. PlaneFlight muffin( "Muffin", "LAX", "Doggie Heaven", 500, 500 );
  2. PlaneFlight sample( "Sample", "LAX", "LAS", 0, 1 );

// I created a PlaneFlight Object of bad data to make sure all my setter and accessor member methods were working as intended:

* 1. PlaneFlight BadData("", "78@$", "78@$", -3, 0);

assert( BadData.getFromCity( ) == "78@$" );

assert( BadData.getToCity( ) == "" );

assert( BadData.getName( ) == "" );

assert( std::to\_string( BadData.getCost( ) ) == "-1.000000" );

assert( std::to\_string( BadData.getMileage( ) ) == "-1.000000" );

// I created a FrequentFlyerAccount to pass in my BadData PlaneFlight object

// Thee initial asserts are just confirming the FrequentFlyerAccounts’ member methods are working well:

* 1. FrequentFlyerAccount BadDataAccount("Bad");

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

assert( BadDataAccount.getName( ) == "Bad" );

assert( BadDataAccount.canEarnFreeFlight( -10 ) == false );

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

// We modified a PlaneFlight object BadData through the freeFlight function because there is enough mileage balance in the FrequentFlyerAccount Muffine:

* 1. FrequentFlyerAccount Muffine("Muffin");

assert( Muffine.freeFlight( "LAS", "LAX", 0, sample ) == false );

assert( Muffine.addFlightToAccount( muffin ) == true );

assert( Muffine.freeFlight( "LAS", "LAX", 100, BadData ) == true );

assert( std::to\_string( Muffine.getBalance( ) ) == "400.000000" );

// I wanted to check what would happen if there was potential conflict when we set the new to/from using freeFlight, if the original to/from coincided with the parameters passed in to freeFlight. I wrote a couple lines of code in my assignment that resets the to and from when freeFlight is called, so there is no conflict when we try to set the new to or from to an existing to or from. For example, the new to/from is LAS/LAX and old to/from is LAX/LAS. Calling flight.setToCity(to) wouldn’t allow me to update the old to because our new to is LAX and that is == to our old from—LAX—which hadn’t been changed yet. That’s why I included the reset calls for to/from when calling freeFlight, so an instance like this could work:

* 1. PlaneFlight EdgeCase("Edge", "LAX", "LAS", 100, 100);

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

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

assert( EdgeCase.getName( ) == "Edge" );

assert( std::to\_string( EdgeCase.getCost( ) ) == "100.000000" );

assert( std::to\_string( EdgeCase.getMileage( ) ) == "100.000000" );

FrequentFlyerAccount EdgeCase1("Edge");

assert( std::to\_string( EdgeCase1.getBalance( ) ) == "0.000000" );

assert( EdgeCase1.addFlightToAccount( EdgeCase ) == true );

assert( std::to\_string( EdgeCase1.getBalance( ) ) == "100.000000" );

assert( EdgeCase1.getName( ) == "Edge" );

assert( EdgeCase1.canEarnFreeFlight( 100 ) == true );

assert( EdgeCase1.freeFlight( "LAX", "LAX", 100, EdgeCase ) == false );

assert( EdgeCase1.freeFlight( "LAS", "LAX", 100, EdgeCase ) == true );

assert( EdgeCase1.freeFlight( "LAS", "LAX", 100, EdgeCase ) == false );

assert( EdgeCase.getName( ) == "Edge" );

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

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

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

assert( std::to\_string( EdgeCase.getMileage( ) ) == "100.000000" );

assert( std::to\_string( EdgeCase1.getBalance( ) ) == "0.000000" );