

NAME: SCOTT PARK UID: 705844713

**CLASSES, FUNCTIONS, AND DESIGN:**

The first class which I created was a class called BloodType. Its name is pretty self explanatory, but it has the parameters of the **int** *EmployeeID* and **char** *type*. It was in this class which I demonstrated the use of both constructor overloading, and constructor delegation. Firstly, to demonstrate Constructor Overloading, I had three declarations of the constructor BloodType, one with no parameters, one with the parameter of **int** *EmployeeID*, and the last with the two parameters of **int** *EmployeeID* and **char** *type*. Thus, depending on the parameter input when testing the constructors/functions, it would call the constructor which the parameters matched. Then came the use of constructor delegation, which I used by delegating the constructor containing the two parameters of **int** *EmployeeID* and **char** *type*, to be delegated to the single parameter constructor, as shown below…

BloodType::BloodType(int *EmployeeID*, char *type*) : BloodType::BloodType( *EmployeeID*)

Now we can look into the functions I implemented which are contained in the BloodType Class. The first function was a setter for the mType, called setType and was a void function which enforced the data validation rules for a possible blood type. In this case, the only possible blood types were that of ‘a’, ‘b’, and ‘o’, and if the blood type wasn’t provided or was an invalid blood type, we would declare the mType as ‘u’ - or unknown. Though it is just a setter, this function holds similarities to other setter functions which were used in BloodDonation and VacationAccount, but with new rules for validations and the use of char. While coding this function, I did not come across any problems or obstacles which I found difficult

The next function within BloodType was a function returning a bool, called bonusType. The logic behind this function was for an employee to be given a monetary bonus for donating, ONLY if their blood type was of type ‘o’. The business logic behind this would be the lack of type ‘o’ blood in blood banks, therefore blood banks incentivize companies, who then incentivize employees to donate blood, especially if they have blood type of ‘o’ ( the universal donor blood). This function takes a BloodType donation, and first checks to see if the given bloodType matches the mType which is stored in the private database. Then it checks for a validID, and finally checks to see if the blood type is of ‘o’. If it passes the validation checks, and is of ‘o’ type, we not only add 100 to the mType private variable, but also call the function to return true. We can find the bonus value through our added getter function, getType which returns mType.

The other class I created was called Vampire. The Vampire class takes in the parameters of type **char** *type*, similar to that of BloodType, but doesn’t need the EmployeeID as a parameter. The function which I implemented into the Vampire class is a bool function called possibleVampire. This function takes in a parameter of the BloodType Class, makes sure the mType and given type match, and then makes use of the ‘u’ type (unknown). If the blood type of the employee happens to be unknown, the function will return true (Assuming other validations previously stated are passed). You might ask yourself – *“what kind of business logic is this?”* – which is a question that lacks the extended consideration of the possibility of a world filled with vampires. We can look at this from a perspective where vampires DO exist in this world. Vampires do not have blood, therefore their blood type is unknown, so if presented with blood type ‘u’, then this function keeps track of whether the bloodType presents the possibility of the employee being a vampire.

**TEST CASES:**

*BloodType unknown(123456);*

*BloodType typeA(123456, 'a');*

*BloodType typeB(654321, 'b');*

*BloodType typeO(123456, 'o');*

*BloodType invalid(12345, 'o');*

*Vampire typeU('u');*

*Vampire typeAA('a');*

assert(unknown.getType() == 'u');

// tests when BloodType is given 1 parameter, mType should be set to u ( unknown )

assert(unknown.getBonus() == 0);

// tests getBonus when the blood type is unknown, there should be no bonus

assert(unknown.bonusType(typeA) == false);

// tests bonusType with a blood type of a, so a bonus is not received and we return false

assert(unknown.bonusType(typeB) == false);

// tests bonusType with a blood type of b, so a bonus is not received and we return false

assert(unknown.bonusType(typeO) == true);

// tests bonusType with a blood type of o, so a bonus IS received and we return true

assert(typeB.getBonus() == 0.000000);

// testing getBonus with blood type b, so mBonus should equal 0

assert(typeO.bonusType(typeO) == true);

// a blood type of o is a valid bonus blood type, so we return true

assert(typeO.getBonus() == 100.000000);

// Again, blood type of o received a bonus, so mBonus received 100 ($)

assert(invalid.bonusType(typeO) == false);

// If the ID doesn't match the stored mID, we return false, even with blood type o

assert(typeU.possibleVampire(typeO) == false );

// type o for blood is NOT the type for a possible vampire, so we return false

assert(typeU.possibleVampire(typeA) == false);

// type a for blood is NOT the type for a possible vampire, so we return false

assert(typeU.possibleVampire(unknown) == true);

// type u for blood IS the type for a possible vampire, so we return true

assert(typeU.getVampire() == true);

// testing getVampire for type U blood type, therefore the stored mVampire should equal true

assert(typeAA.possibleVampire(unknown) == false);

// testing possibleVampire where mType is not equal to the given type, so we return false

assert(typeAA.getVampire() == false);

// testing getVampire for type a blood type, therefore the stores mVampire should equal false