1. Classes, Interfaces and Types

25% of Midterm score

Code Fragment	A/M/F	Code Fragment	A/M/F
`motorcycle.goFast();`	**(A) A**	`vehicle = new Pickup();`	**(F) A**
`dragon.goFast();`	**(B) A**	`fourWV.go()`	**(G) A**
`sportyConveyance.goFast();`	**(C) A**	`vehicle = new Dragon();`	**(H) F**
`vehicle.goFast();`	**(D) F**	`car = fourWV;`	**(I) F**
`((Vehicle) sportyConveyance).go();`	**(E) M**	`vehicle.turnOnRadio();`	**(J) F**
`motorcycle = (Motorcycle) sportyConveyance;`			**(K) M**
`((SportyConveyance) vehicle).goFast();`			**(L) M**
`((SportyConveyance) vehicle).turnOnRadio();`			**(M) M**
`((FourWheeledVehicle) sportyConveyance).turnOnRadio()`			**(N) M**

2. Creating Classes and Methods

25% of Midterm score

Assume the following partial class/interface definitions for **Bank, Account,** and **Customer.** *Read all the code before you begin!*

a) Fill in all the methods marked **TODO** .(60% of question 2)

```
public class Bank {
    private final String name;
    private final List<Account> accounts;

       public Bank(String name) {
         this.name = name
         accounts = new ArrayList<Account>();
     }

    public void addAccount(Account a){ accounts.add(a); }

    public int getTotalDeposits() {
        int result = 0;
        for (Account a : accounts) {
            result += a.getBalance();
        }
        return result;
    }

    public int getNumberOfAccounts() {
        return accounts.size()
    }
```

```
    public String toString() {
        return name + " (" + getNumberOfAccounts() + " accounts)";
    }
}

public class Account {
    private int balance;      // Account balance in cents
    private Customer owner;

    public Account(int balance, Customer owner) {
        this.balance = balance;
        this.owner = owner;
    }

    public Customer getOwner() {
        return owner;
    }

    public int getBalance() {
        return balance;
    }

    public int addToBalance(final int amount) {
        balance += amount;
    }

    public int subtractFromBalance(final int amount) {
        balance -= amount;
    }
}
```

(2 continued) Answer the following questions using the above classes.

b. Fill in the method below so that it creates and returns a **Bank** object with two accounts, each with a zero balance and a different owner. Use any names you like for the bank and the owners. You may assume the method is part of a class other than **Bank**; the containing class is not shown. (30% of question 2)

```
public Bank createTestBank() {

    Bank bank = new Bank("Bank Of Oz");
    bank.addAccount(new Account(0, new Customer("Bruce")));
    bank.addAccount(new Account(0, new Customer("Sheila")));
    return bank;
}
```

c. Add a method to the **Bank** class to charge an account maintenance fee to all accounts. It's OK for a balance to become negative. The fee is given in cents. You may assume the method is part of the **Bank** class; the containing class is not shown. (10% of question 2)

```
public void chargeMaintenanceFee(final int fee)
{
     for (Account a : accounts) {
         a.subtractFromBalance(fee);
     }
}
```

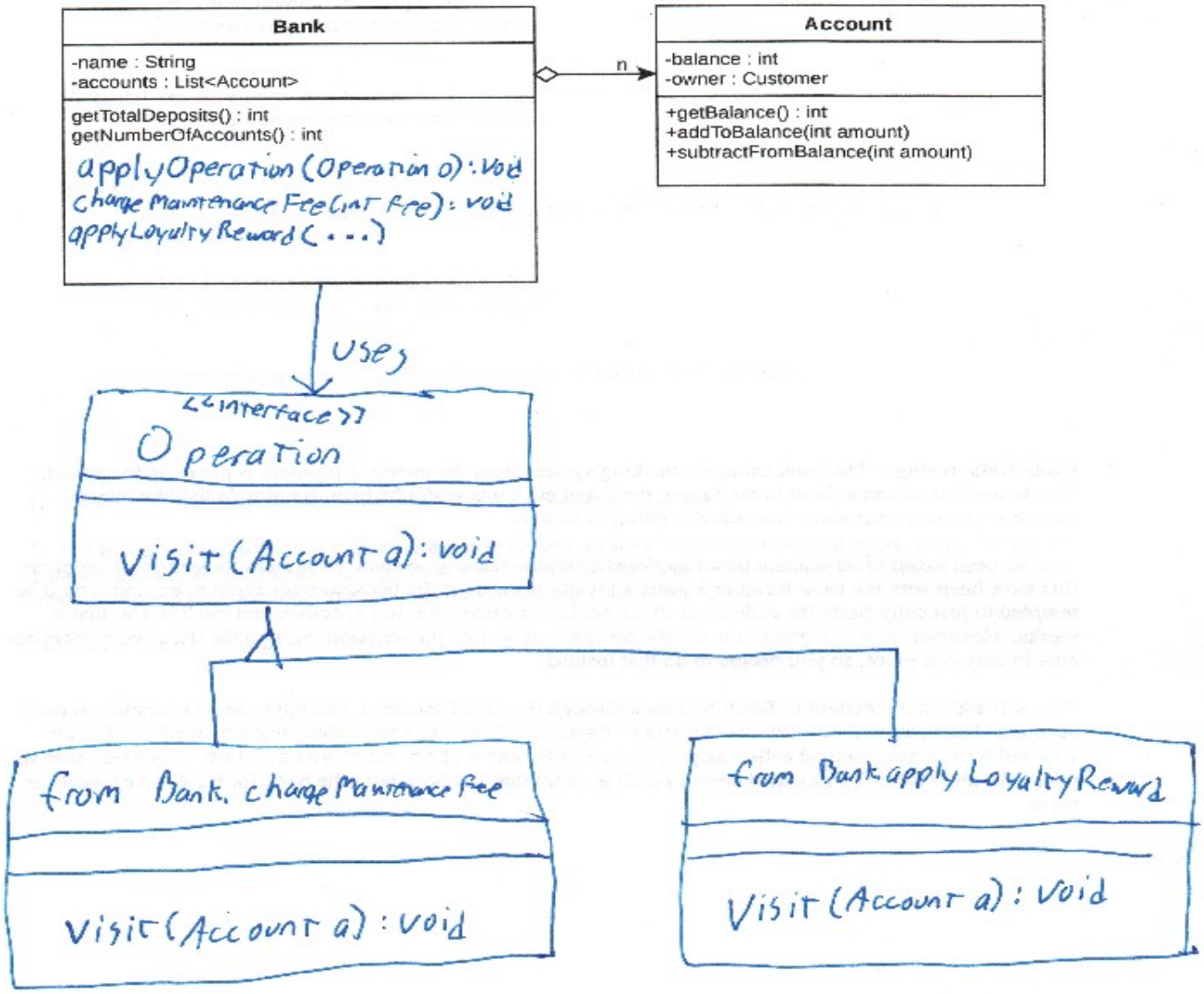
3. **Code Refactoring** – The bank using the banking system from the previous problem is planning to expand. You know that, at some point in the future, the list of accounts won't be kept in a simple list like they are now, but you don't yet know how they're going to be kept.

   You've been asked to implement new functionality on the banking system: A system for giving all accounts that have been with the bank for over 5 years a loyalty reward. If the bank weren't about to expand, you'd be tempted to just copy-paste the code from `chargeMaintenanceFee()`, above, and modify it so that it works. However, now is a good time to refactor the code so that the iteration through the list of customers occurs in only one place, so you decide to do that instead.

   You will add a new method to **Bank** to iterate through the list of accounts, and apply some operation to each account. Then you will re-write `chargeMaintenanceFee()` so that it uses your new method. Finally, you will write a new method called `applyLoyaltyReward(int minYears, int reward)` that applies a loyalty reward of `reward` cents to each account that has been with the bank for `minYears` years or more.

(25% of midterm total)

a. complete the following UML diagram so that it includes any classes, interfaces or methods you have added or modified as part of your refactoring. Any anonymous classes may be shown with the name "from XXX" where "XXX" indicates where the class is defined. Solid lines and non-italic text are fine.



(50% of question 3)

(b and c combined are 50% of question 3)

b. Give the code for any new named classes or interfaces you introduce in your refactoring here:

```
public interface Operation {
    public void visit(Account a);
}
```

c. Fill in the implementation of the following methods of **Bank**. You may assume these methods are part of the **Bank** class; the containing class is not shown.

```
public void applyOperation(Operation o) {
    for (Account a : accounts) {
        o.visit(a);
    }
}

public void chargeMaintenanceFee(final int fee) {
    applyOperation(a -> { a.subtractFromBalance(fee); });
}

public void applyLoyaltyReward(final int minYears, final int reward)
    applyOperation(a -> {
        if (a.getOwner().getYearsAtBank() >= minYears) {
            a.addToBalance(reward);
        }
    });
}
```

3. `Object.equals()` and `Object.hashCode()` (25% of midterm total, evenly split betwwen equals() and hashCode()l)

Complete the following class:

```
public final class PersonName {

    public final String firstName;   // guaranteed to not be null
    public final String lastName;    // guaranteed to not be null

    public PersonName(String firstName, String lastName) {
        // implementation not shown
    }

    public boolean equals(Object other) {
        // TODO:  Fill in code here

        if (other instanceof PersonName) {
            PersonName pn = (PersonName) other;
            return firstName.equals(pn.firstName)
                   && lastName.equals(pn.lastName);
        } else {
            return false;
        }
    }
```

```
    /**
     * Returns a value consistent with the definition of equals()
     * such that PersonName instances can be used as the key in
     * a hash table.
     */
    public int hashCode() {
        return firstName.hashCode()*31 + lastName.hashCode();
    }
```

4. Extra Credit (4%, no partial credit): Advanced Refactoring

Rewrite the method `getTotalDeposits()` of **Bank** so that it uses `applyAccountOperation()` to calculate the total.

a. Give the code for any new named classes or interfaces you introduce in your refactoring here:

Solution 1:

```
    public class TotalDepositOperation implements Operation {

        public int total = 0;

        @Override
        public visit(Account a) {
            total += a.getBalance();
        }
    }
```

Solution 2:

```
    No new named classes or interfaces
```

b. Fill in the implementation of the following method of **Bank**. You may assume it is part of the **Bank** class; the containing class is not shown.

Solution 1:

```
public int getTotalDeposits() {
    TotalDepositOperation o = new TotalDepositOperation();
    applyOperation(o);
    return o.total;
}
```

Solution 2:

```
public int getTotalDeposits() {
    final int[] total = new int[1];
    applyOperation(a -> { total[0] += a.getBalance(); } );
    return total[0];
}
```