Kyle Reinholt

OOAD Homework1

Questions 1-3

1.
    a. Abstraction – This refers to showing only the necessary parts of your data to the developer. In comparison with encapsulation that focuses on hiding data, abstraction is about showing specific components of your data. An example of this is making a class method public. If we have class with 4 members or variables and 5 methods. We may want to make some of those members and variables public. A **good** implementation of abstraction would only make class members and methods public if necessary. For example, another class needs to access those methods or members. Otherwise, you should not use abstraction because the more public members and methods, the more places you must look when debugging. In comparison to encapsulation, if you must debug an issue that was traced back to a private variable, you know only the class itself can manipulate that data, making your job of tracking it down much easier than in the public case. A **bad** example of abstraction would be making members and methods public that don't need to be. If something goes wrong, tracking down the issues could be difficult.
    b. Encapsulation – As mentioned above, encapsulation is used to describe hiding components of your data. An example of this is marking class members as private. A **bad** example of encapsulation would consist of making every member and method of a class private. This would render the class useless by any other class in the program. A **good** example of encapsulation is making members and methods private that do not need to be used by other classes in the program.
    c. Cohesion – Cohesion is related to coupling (defined below). This term explains the interdependence of instructions within a routine. This is a hard term to describe in less than a paragraph. For a short description, if you can change one instruction in a routine and the change correctly propagates throughout the rest of the instructions and the result of the routine is correct, you are likely to have tight cohesion (this is **good**). If you change one instruction and then you must implement a change in all the other instructions manually, you have loose cohesion, this is **bad**.
    d. Coupling – Refers to the strength of the connection between routines. For each routine or class, we want one connection bonding them. If a routine is connected to many other routines, any change made in this routine will propagate to all the other routines and debugging will take a long time (this is **bad**). Compared to a loosely coupled connections, the routine with one connection only affects one routine, thus making debugging much more efficient if changes cause problems (this is **good**).

2. Payroll System – Functional Decomposition Approach
   This is how I would approach this problem at a very high level. Since all of the information is in the database, we can use select statements to get hours worked, hourly pay rate, and banking information. Our program needs to retrieve that data and complete a deposit transaction with each employee's bank account.
    a. Connect to database
    b. For each employee_id in the database

i. Select hours worked over monthly pay period
ii. Select hourly rate for ith employee
iii. Compute Monthly pay
iv. Select ith employee's routing number and account number
v. Connect to ith employee's bank server
vi. Complete transaction by inserting monthly payment amount into the the ith employee's account
vii. Close connection to bank server

c. Close connection to database

3. Payroll System – Object Oriented Approach
    a. Objects
        i. **DatabaseHandler** – This object will connect to the database commit queries on the database. It will essentially retrieve all of the information we need to pay the employees.
            1. **EmployeeList**
            2. **Functionality**
                a. getListOfEmployees()
                b. getHoursWorked(Employee,Month)
                c. getPayRate(Employee)
                d. computeMonthlyPay(hoursWorked, payrate)
        ii. **Payroll** – This object will connect to each employees bank server and deposit their monthly pay into the account.
            1. **DatabaseHandler**
            2. **Functionality**
                a. payEmployees()
                b. pay(Employee,wage)
        iii. Employee
            1. Id
            2. First_name
            3. Last_name
            4. Address
            5. routing_number
            6. account_number
            7. Functionality
                a. getRoutingNumber()
                b. setRoutingNumber(routingNum)
                c. getAccountNumber()
                d. setAccountNumber(accountNum)
                e. getId()
                f. setId(id)
        iv. EmployeeList
            1. Functionality
                a. addEmployee(Employee)

b. Order of instantiation – This is what main would look like
    i. **Payroll** payroll = new **Payroll**();
    ii. payroll.payEmployees();
c. For a further breakdown of what goes on in **Payroll**, see d.
d. payroll.payEmployees() broken down
    i. **DatabaseHandler** DBHandler = new DBHandler();
    ii. List**<Employee>** workers = DBHandler.getListOfEmployees();
    iii. Foreach **Employee** in workers:
        1. hoursWorked = DBHandler.getHoursWorked(Employee);
        2. payrate = DBHandler.getPayrate(Employee);
        3. wages = DBHandler.computeMonthlyPay(hoursWorked,payrate)
        4. pay(Employee,wages)