CSCI 5448 Project 1

Michael Voecks, ID: 102584009

Question 1: Term Definition

- **Abstraction:** The process of hiding irrelevant information while allowing access to only the necessary components of a system. This makes the system easier to use as the user only has to worry about what it does, and not how it works. In class design, abstraction allows us to create many different classes that serve independent purposes while simultaneously allowing for easy communication between different class types.
  - **Good Example in Class Design** - Creating an abstract class that represents an overarching category, such as Animals, with abstract methods and some standard normal methods that all Animals will exhibit. Then, creating subclasses of the main class that extend upon the already built methods and complete the abstract methods left available by the main class.
  - **Bad Example in Class Design** - Creating an abstract class with no abstract methods.
- **Encapsulation:** The practice of only allowing an object to edit its own functions and variables, enabling the object to interact with the main program and other objects with a lower likelihood of causing errors. Within a given programs class, encapsulation would include keeping all local variables private while having public getters and setters to edit them, and also allowing for protected variables/functions that can only be accessed by objects of the same class.
  - **Good Example in Class Design** - Keeping variables of a class private and only making getters/setters for the ones needed to be changed by other classes. In addition we will only allow for public functions when necessary for outside objects to use them.
  - **Bad Example in Class Design** - Creating a class with all methods and variables that are public and can be edited by any other class or function.
- **Polymorphism:** Allowing objects to be redefined into sub-objects that inherit the properties of the super-class, while also being changed to fit a different purpose. In class design this allows us to create a hierarchy of classes that inherit properties (through abstract classes/interfaces) from their parent class to reduce the amount of redundant code that is put in the final program. In addition this standardizes the way in which objects interact with each other to simplify the interactions between objects.
  - **Good Example in Class Design** - Creating a set of classes that inherit the same methods from a higher abstract class, and then implementing the specifics of how each object performs those methods withing the objects class itself. Then, when objects need to interact with one another we know exactly the what the end result should look like but the details of doing so are left to each objects definitions.
  - **Bad Example in Class Design** - Creating a series of classes that need to communicate with each other using no standard definition of the methods in each class. I.e. Programming a dog and cat class that don't share any common methods but must "talk" with one another.
- **Cohesion:** Cohesion is a measure of the modularity of objects within a program. A program that has high cohesion has objects that are programmed for singular purposes, while low cohesion programs have objects that perform many tasks that are loosely related. In class design, we seek high cohesion programs so modules can be used only in situations which they apply to, creating better designed code that is easier to maintain and update.

  - **Good Example in Class Design** - Creating various different objects that serve singular purposes, in the example of a zoo that we see later in Question 4, we want different objects for different animals and the zookeeper, as well as some abstract classes to define the structure of the animals.
  - **Bad Example in Class Design** - Creating one object that will perform every task in the requirements of the code.

- **Coupling:** Coupling is a measure of the interdependence of objects within a program. High coupling implies that the program has many interconnected components and updating one of them will change the behavior of other components, and many objects need to modified to implement even simple changes. However, in class design we seek low coupling which is shown through components of the code being separate entities from one another that can still interact with one another.
  - **Good Example in Class Design** - Consider the example of programming a payroll system from Questions 2-3. We can create a an abstract class Employee that allows for different types of employee and requires them to implement how much they should be paid given their hours worked, and a different class Boss that pays each Employee ever month. This is a loosly coupled system that allows for easy modification to allow for different types of employees and pay structures.
  - **Bad Example in Class Design** - Using the same example, If we create a class that only reports the number of hours worked and a second class that summed the hours worked and paid accordingly, we have a tightly coupled system. That is if we want to change the way a specific employee is paid, i.e. salaried vs hourly, we would need to modify both classes to make the change.

Question 2: Payroll System, Functional Approach

Functional Decomposition

1. Connect to the database
2. Query the data for each employee, retrieving the (employee name), (# hours worked this month), and their (Pay Rate)
3. Loop through each employee:
   (a) Convert (Pay Rate) to (Hourly Wage)
   (b) Multiply (Hourly Wage) by (# hours worked this month) to determine how much they should be paid
   (c) Pay the employee this amount electronically
4. Close connection with database.

Assumptions

- The database contains the correct info for amount of hours worked this month, the pay rate, the employee type, and the name of each employee that needs to be paid.
- For the purpose of an example, this company only has three types of employee, each type being paid the same amount, each described below:
  - Supervisors - Not paid hourly, rate of pay stored in the database is stored as a monthly wage
  - Customer Service - Paid hourly
- Employees who have a pay structure that is different from an hourly wage can have their monthly pay converted to an hourly rate accurately.
- The program can pay the employee easily and electronically if given only the employee name and amount to pay.through the payEmployeeElectronically(name, payAmmount) function.

Design Pseudocode

Algorithm 1 Functional Payroll System

```
1: procedure payEmployees()
2:     EmployeeDatabase ← connectToEmployeeDatabase()        ▷ Connect to the Database
3:     employeeData ← Gets array of employee data from EmployeeDatabase
4:     loop over each employee E:                            ▷ Loop over each employee's data
5:     if E(type) = 'Supervisor' then                        ▷ Determine the type of employee
6:         pay ← E(payRate).
7:     else
8:         pay ← E(payRate) * E(hoursWorked)
9:     payEmployeeElectronically(E(name), pay)               ▷ Pay all the employees
10:    closeConnection(EmployeeDatabase)                     ▷ Close connection with database
```

Question 3: Payroll System. Object Oriented Approach

Objected Oriented Planning

1. Have one abstract class representing Employees. This abstract class has private variables (Name), (Pay Rate) and (Hours Worked), has an abstract function calculatePaystub() as well appropriate getters/setters. The Employee class will have two subclasses:
   - Supervisors - Has a calculatePaystub() function that doesnt rely on hourly wage, and instead returns just the Pay Rate
   - CustomerService - Has a calculatePaystub() function that multiplies their payRate by the number of hours worked.
2. A second class Timekeeper is used to keep track of the hours each employee works, and subsequently pay each employee at the end of the month. This class has the private variables (Name), and (myEmployees) which is an array of Employee objects that the Timekeeper is responsible for. It also has two functions, addEmployee(Employee E) which adds employees to pay, and payEmployees() that pays each employee in the myEmployees array.
3. The main class will be a runner that first connects to the database creates objects representing each employee using the employee info in the database. Then, it will initialize a Timekeeper object and give it the employee objects that were created earlier, and this object will pay each employee.

Assumptions

- Each employee is created by passing the arguments from the database into a constructor that initializes the the correct object appropriately, i.e. The database knows to create a Supervisor or Customer service object when creating employees.
- Each Employee subclass has defined calculatePaystub() to pay them the appropriate amount based of the data given from the database to construct it.
- The Timekeeper object is created with no arguments, and accepts Employee objects to add through its addEmployee(Employee E) function.
- A function, createEmployee(employeeData) will return a Employee object of the correct type (Supervisor or CustomerService) based off their data 'employeeData' from the database

Design Pseudocode

Algorithm 2 Object Oriented Payroll System

```
1: procedure MAIN()
2:     ArrayList ⟨Employee⟩ employeeList ← new ArrayList ⟨Employee⟩ [ ]   ▷ Create an array of Employee objects
3:     EmployeeDatabase ← connectToEmployeeDatabase()   ▷ Connect to the database
4:     employeeData ← Gets array of employee data from EmployeeDatabase
5:     loop over each employee E in employeeData:   ▷ Loop over each employee
6:     employeeList.add(createEmployee(E))   ▷ append employee E to employeeList
7:     end loop
8:     Timekeeper bossMan ← new Timekeeper(employeeList)   ▷ Initialize a Timekeeper
9:     bossMan.payEmployees()   ▷ Use the timekeeper to pay each employee
10:    closeConnection(EmployeeDatabase)
```